Lazarus

La Biblia del SynEdit

FICHA TÉCNICA

AUTOR	Tito Hinostroza – Lima Perú
FECHA	Rev5 terminada en 12/04/2014
APLICABLE A	Paquete SynEdit de Lazarus 1.0.12 Los ejemplos se han desarrollado sobre Windows-32.
NVEL DEL DOCUMENTO	Medio. Se asume conocimientos de Free Pascal y Lazarus.
DOCUMENTOS PREVIOS	ninguno
BIBLIOGRAFÍA	Código fuente de SynEdit - Lazarus Código fuente de SynEdit - SourceForge http://forum.lazarus.freepascal.org/ http://wiki.freepascal.org/SynEdit/es http://wiki.freepascal.org/SynEdit

CONTROL DE CAMBIOS

VERSIÓN	FECHA	DESCRIPCIÓN MODIFICACIONES
Rev1	12/10/2013	Por Tito Hinostroza. Primera versión revisada completa de la documentación. Queda pendiente de documentar: • Los otros controles del paquete SynEdit. • Funcionamiento más detallado en modo columna. (smCurrent) • El uso de complementos. • El autocompletado.
Rev2	12/10/2013	Por Tito Hinostroza Se ha corregido la sintaxis de algunos ejemplos. Se amplió la sección 1.4.1 y se corrigió el gráfico. Se agregó información a Sección 1.4.4
Rev3	19/10/2013	Por Tito Hinostroza Se agregó Apéndice y se incluyó información sobre el algoritmo "hash", usado en la implementación de resaltadores en Lazarus. Se modificó sección 2.3.6. Se agregó sección 1.7.2 Se modificó sección 2.4
Rev4	27/10/2013	Por Tito Hinostroza Se pasó la tabla de propiedades al final Se agregó sección 1.4.9 sobre las propiedades "Options" y "Options2" Se agregó información a la sección 1.4.2 y se agregó sección "Tipografía". Se reordenó sección 2 de resaltado de sintaxis y se completó la introducción. Se agregó información sobre más propiedades de SynEdit.
Rev5	26/01/2014	Por Tito Hinostroza Se corrige algunas palabras con errores en la Sección 1.3 Se modificó Sección 1.4.2 Se modificaron y completaron varias secciones. Se creó la sección "Modificar el contenido" Se agregó información sobre la creación de atributos en 2.3.4 Se agrega sección 2.4. Se agrega información adicional referida a plegado de código.
Rev6	05/04/2014	Se corrige errores tipográficos. Se agrega información sobre las clases TSynCustomFoldHighlighter y TSynCustomHighlighter. Se incluyen más propiedades y métodos en la sección 1.9. Se crea la sección 1.5.1

“En el principio, era TECO y VI”

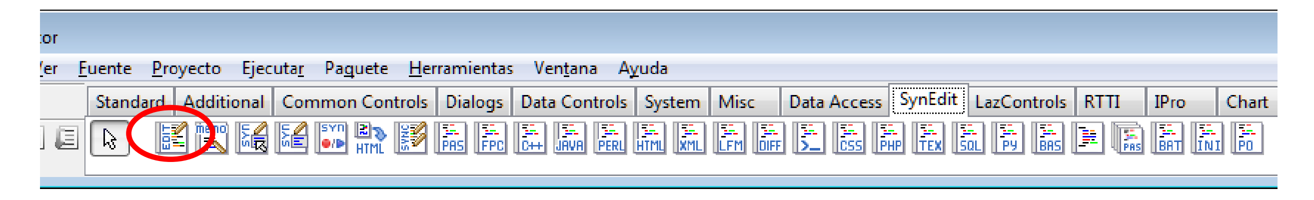
1 Editor con reconocimiento de Sintaxis: SynEdit.

Gran parte de este trabajo está basado en la experiencia de uso, la escasa documentación que existe en la web, Ingeniería inversa, y al análisis del código fuente del componente SynEdit.

1.1 ¿Qué es SynEdit?

Es un componente o control que viene integrado en el entorno de Lazarus. Es un control de edición. Permite implementar rápidamente, editores de texto, con características avanzadas como resaltado de sintaxis.

Para ser exactos, SynEdit es todo un paquete que viene ya integrado en Lazarus cuando se instala (y que incluye diversos componentes), pero por lo general, cuando decimos SynEdit, nos referimos al componente TSynEdit que es el editor con posibilidades de resaltado de sintaxis.



Se puede acceder a él, a través de la pestaña "SynEdit" de la paleta de componentes. EL editor TSynEdit, se encuentra a la izquierda. En esta pestaña hay además diversos controles relacionados a "TSynEdit":

- TSynMemo.- Versión de TSynEdit con algunas diferencias. Tiene menos métodos y eventos publicados. Deriva de SynEdit. Puede remplazar a SynEdit, en muchos casos.
- TSynCompletion.- Control no visible que permite implementar la opción de "Completado de código".
- TSynAutoComplete.- Control no visible que permite implementar la opción de "Auto-Completado de código".
- TSynPasSyn.- Componente de sintaxis del lenguaje Pascal.
- TSynFreePascalSyn.- Componente de sintaxis del lenguaje de Free Pascal.
- TSynCppSyn.- Componente de sintaxis del lenguaje C++.
- TSynJavaSyn.- Componente de sintaxis del lenguaje Java.
- etc.

El control SynEdit, que se incluye en Lazarus, es una versión modificada del proyecto independiente SynEdit. La versión adaptada para Lazarus, se ha desarrollado a partir de la versión 1.03,

a la que se le ha agregado algunas características adicionales, como soporte para UTF-8 y Plegado de código.

Este componente, está bien revisado y comprobado, ya que es el mismo que usa el IDE de Lazarus para su Editor de Código.

Desgraciadamente no existe suficiente documentación técnica sobre el proyecto, pero lo que sí se sabe es que es funcional y de muy buen desempeño.

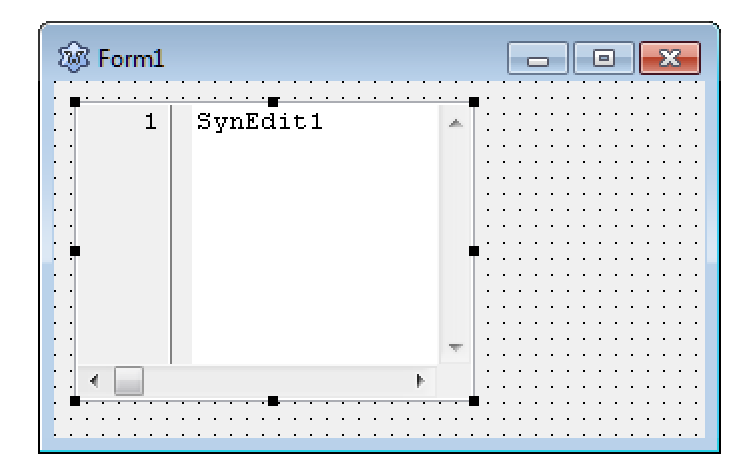
1.2 Características de SynEdit

El componente SynEdit (TSynEdit) de Lazarus, tiene las siguientes características:

- Componente accesible desde la IDE Lazarus.
- No requiere archivos, adicionales (como es el caso de Scintilla). Una vez integrado en el proyecto, se integra en el código sin ninguna dependencia.
- Su código es completamente accesible y modificable.
- Trabaja completamente en la codificación UTF-8.
- Soporta coloreado de sintaxis, para varios lenguajes predefinidos o se le puede crear una nueva sintaxis.
- Soporta opciones de completado y autocompletado de código.
- Soporta plegado de código (folding). Pero debe hacerse por código.
- Incluye opciones de "Undo" y "Redo", con amplia memoria de cambios.
- Contiene métodos para Búsqueda y Reemplazo.
- Soporta selección sencilla por columnas.
- Permite numerar las líneas.
- Soporta resaltadores y marcadores de texto.

1.3 Apariencia

Al agregar el componente TSynEdit al formulario, este se encuentra ya operativo. Se puede ejecutar el programa y ver que el editor responde como cualquier cuadro de texto de tipo TMemo.



La principal diferencia visual está en la barra vertical que aparece a la izquierda. Esta barra sirve para mostrar el número de línea, y para otras opciones más. Otra diferencia es que el tamaño horizontal de letra es uniforme. Es decir que la letra "m", tiene el mismo ancho que la letra "l". Este es el tipo de letra que se carga por defecto en un "SynEdit".

Inicialmente el SynEdit no incluye opciones de resaltado de sintaxis, porque no tiene ninguna sintaxis asociada, todavía. Lo que si incluye por defecto, es la detección de "brackets", es decir resalta el paréntesis que se abre y se cierra, si es que se pone el cursor en uno de los paréntesis. El comportamiento es similar con los corchetes, llaves y comillas. No se reconocen los apóstrofos.

Este resaltado consiste, por defecto, en poner los caracteres inicial y final, en modo negrita.

```
texto (texto entre paréntesis (otro texto)) más texto.
```

Para desactivar esta característica, se debe quitar la opción "eoBracketHighlight", de la propiedad "Options".

Si se desea modificar el atributo del resaltado de los delimitadores, se puede usar el siguiente código:

```
SynEdit1.BracketMatchColor.Foreground := clRed;  //cambia a color rojo
```

Otra de las características que viene por defecto en SynEdit, es la opción de poder crear marcadores (Ver 1.8.5 - Marcadores de texto). SI no se va a usar, esta opción se debe deshabilitarla porque podría generar errores en tiempo de ejecución.

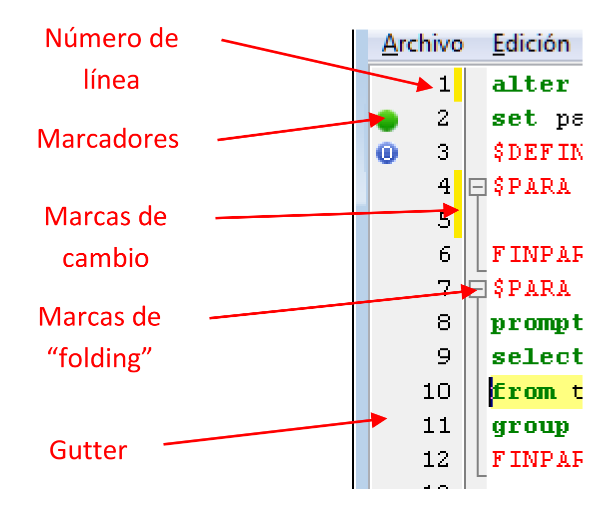
También las opciones de Cortar, Copiar y Pegar, se encuentran habilitadas por defecto, en el control SynEdit, sin necesidad de implementarlas.

En general, todos los atajos que se crean por defecto en SynEdit, corresponden a acciones que están predefinidas sin necesidad de activarlas.

Existen diversas propiedades para cambiar la apariencia del control SynEdit. Describiremos algunas de ellas.

1.3.1 Panel Vertical

El panel vertical que aparece en la parte izquierda del control, es llamada "Gutter" (canal), y es la destinada a mostrar el número de línea, las marcas de plegado (folding), las marcas de cambio y los marcadores.



El "Gutter", se puede mostrar u ocultar por código. Para hacerlo invisible se debería hacer:

```
SynEdit1.Gutter.Visible := False;
```

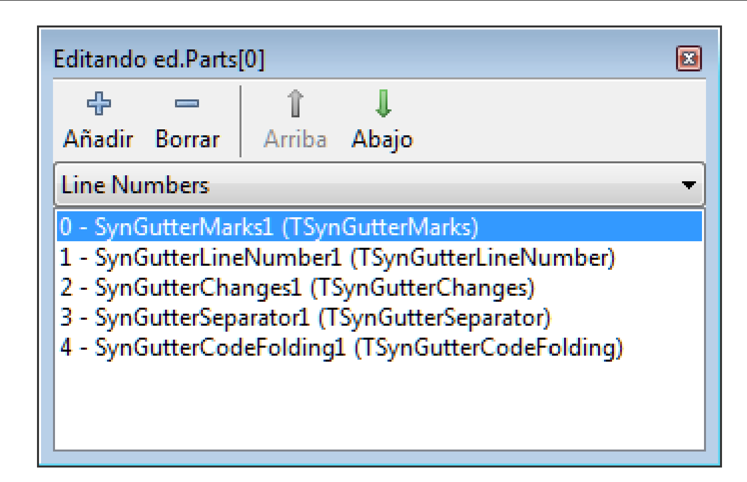
En este caso, nuestro editor tiene el nombre por defecto que se le asigna al agregarlo a un formulario: SynEdit1.

El "Gutter", por defecto, tiene el ancho ajustado de forma automática, es decir que cambia, de acuerdo a la cantidad de filas en el editor. Se le puede fijar a un ancho determinado, poniendo primero la propiedad "Autosize" a "false":

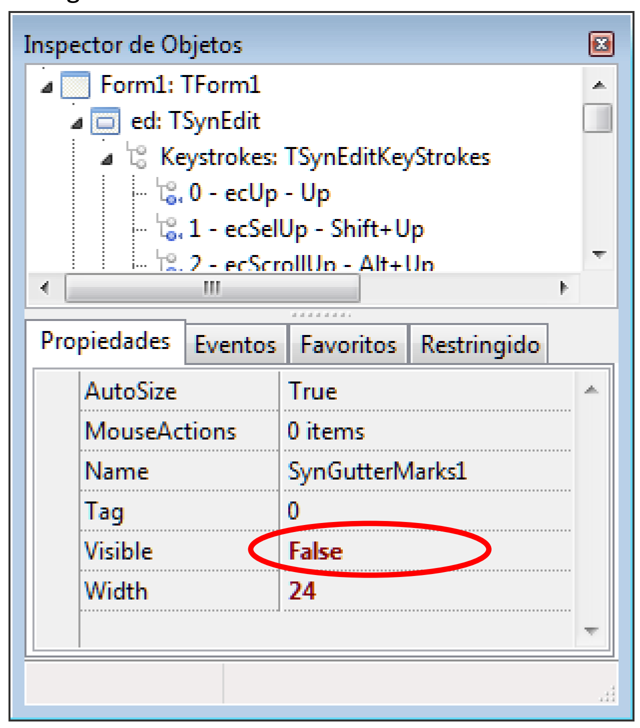
```
ed.Gutter.AutoSize:=false;
ed.Gutter.Width:=30;
```

No es recomendable cambiar así, el ancho, porque de esta forma, no se reubican los elementos que contiene, así que se podría perder de vista parte de los números o las marcas de "folding".

Es preferible dejar el "Autosize" en "true" y desactivar elementos individuales del "Gutter", para variar su tamaño. Esto se puede hacer fácilmente con el inspector de objetos, modificando la propiedad "Parts", de la propiedad "Gutter":



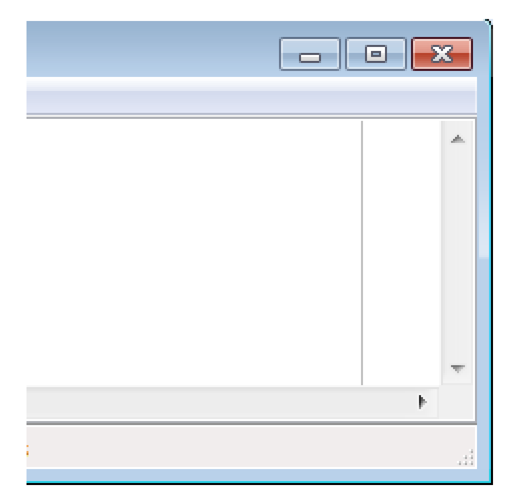
Y luego ocultando el elemento deseado:



En este ejemplo se oculta el área destinada a los marcadores. Al ir ocultando áreas, el tamaño total del “Gutter”, va disminuyendo.

1.3.2 Margen Derecho

Por defecto en SynEdit aparece una línea vertical en la parte derecha del texto, usualmente en la columna 80. Esta línea sirve de ayuda para cuando se quiera imprimir el contenido y se desee evitar sobrepasarse en el tamaño de la línea que permite la impresora.



Para cambiar la posición se debe modificar la propiedad “RightEdge”:

editor.RightEdge:= 100;   //fija posición de la línea vertical

También se puede cambiar su color usando la propiedad “RightEdgeColor”.

Si no se desea que esta línea aparezca, se puede poner su posición en una coordenada negativa:

```
editor.RightEdge:= -1;   //oculta línea vertical
```

O se puede desactivar usando la opción “eoHideRightMargin”, de la propiedad “Options”:

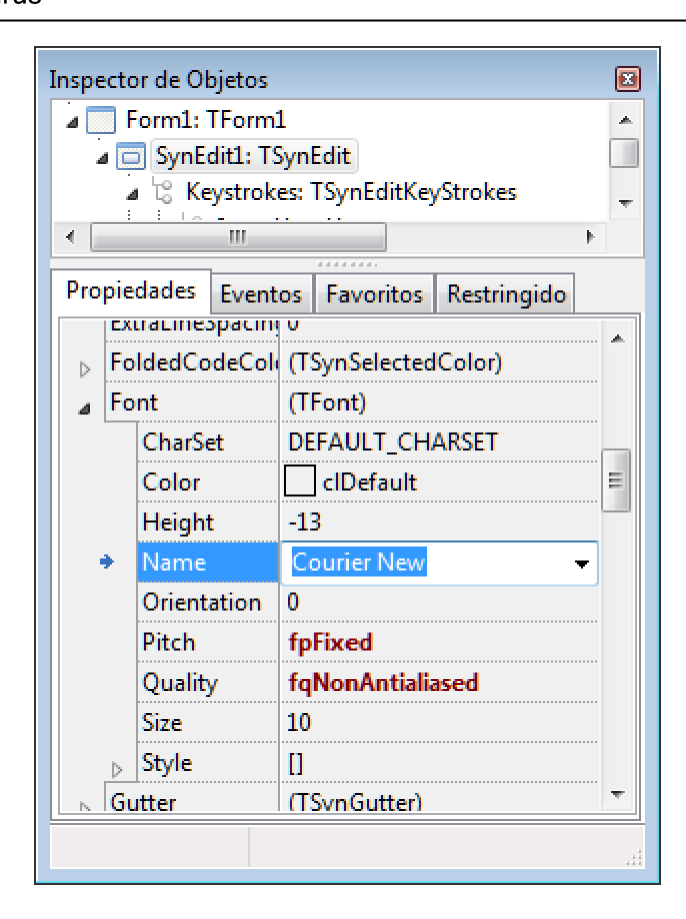
```
editor.Options := editor.Options + [eoHideRightMargin];
```

Las barras de desplazamiento del editor se pueden ocultar o mostrar usando la propiedad “ScrollBars”.

1.3.3 Tipografía

SynEdit permite configurar diversas propiedades de la tipografía a usar. Por defecto el texto se muestra con la fuente “Courirer New” en tamaño 10.

Para cambia la fuente que usará sin SynEdit se debe configurar el objeto Font. Esta tarea se puede hacer por código o usando el inspector de objetos:

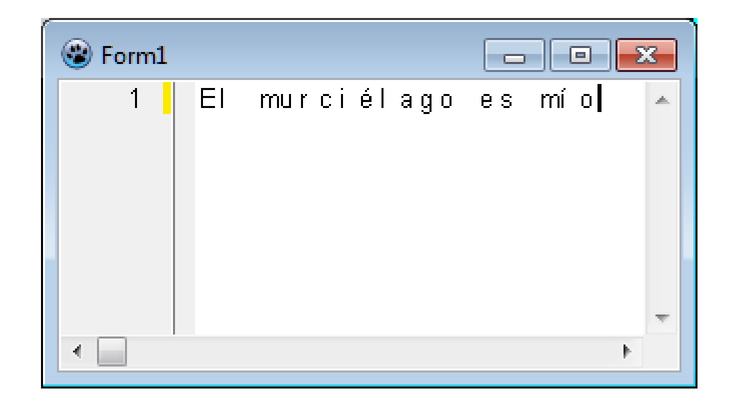


El objeto Font, tiene diversas propiedades y métodos, muchos de los cuales, solo pueden ser accedidos por código.

Quizá la propiedad más común, para cambiar la apariencia del texto en pantalla es la fuente. Esta se puede cambiar mediante la propiedad "Name".

Considerar que los caracteres a mostrar en SynEdit son siempre monoespaciados, es decir que todos los caracteres tendrán el mismo ancho en la pantalla. Si se usará una fuente con ancho distinto para cada carácter, SynEdit lo mostrará igual como una fuente monoespaciada, dando la impresión de que los caracteres no están igualmente espaciados.

El siguiente ejemplo muestra un editor en el que se ha usado el tipo de letra "Arial", que no es monoespaciada:

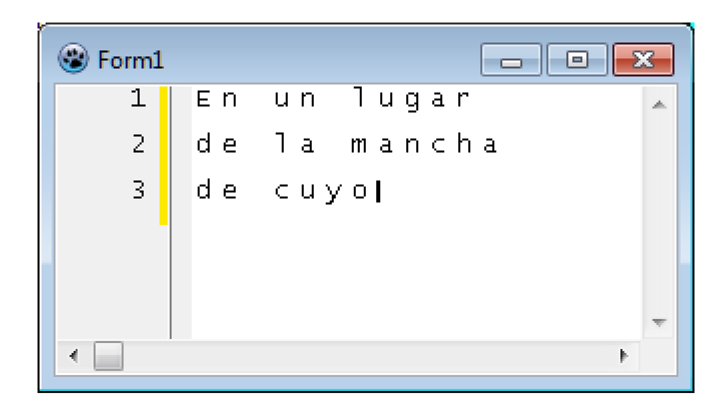


El texto ha sido correctamente escrito y separado, pero como las letras en "Arial", tienen ancho distinto (la "m" es más ancha que la "i"), el texto da la apriencia de estar mal espaciado. Debido a este

efecto, se recomienda usar solo los tipos de letra que son de ancho uniforme, como Courier, Fixed, o Lucida.

El tamaño de la fuente se define modificando la propiedad “Size” y el color con la propiedad “color”. La propiedad “Style”, permite definir los atributos: negrita, subrayado y cursiva.

También es posible cambiar el espaciado entre líneas y entre caracteres, usando las propiedades “ExtraCharSpacing” y “ExtraLineSpacing”:



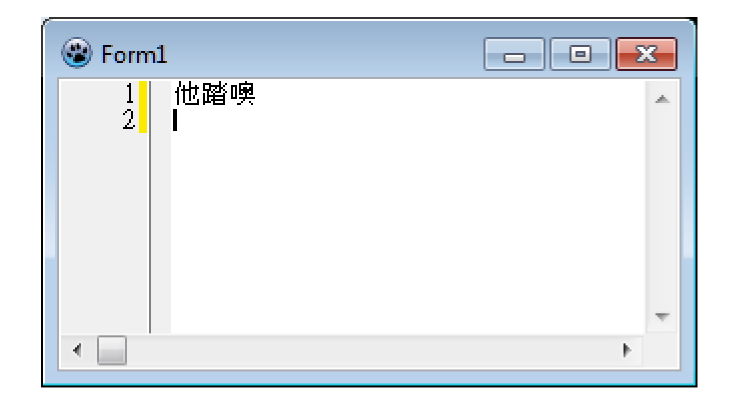
La figura anterior se ha obtenido, usando el siguiente código:

```
SynEdit1.ExtraCharSpacing:=5;
SynEdit1.ExtraLineSpacing:=10;
```

Por defecto el espaciado es cero. Si se desea juntar, en vez de separar, se pueden usar valores negativos para estas propiedades.

Otra de las propiedades que podemos usar para personalizar a SynEdit, es el juego de caracteres (CharSet).

El juego de caracteres permite cambiar el idioma a usar en el editor. Por ejemplo, usando el juego de caracteres CHINESEBIG5_CHARSET, podríamos usar caracteres chinos:

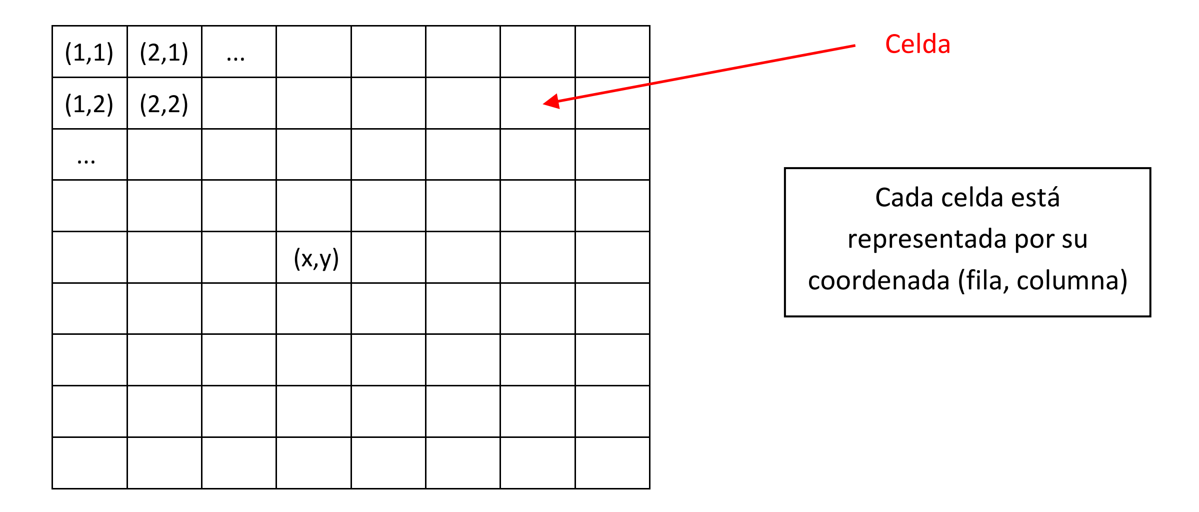


Algunos juegos de caracteres, como el del ejemplo, usan para un caracter, el doble de espacio en el editor, que los caracteres tradicionales occidentales. No hay problema, SynEdit, puede manejar este tipo de caracteres e inclusive combinar caracteres de espacio simple y doble (o de dos juegos distintos de caracteres) en el mismo documento.

1.4 Funcionamiento

1.4.1 Coordenadas del editor

Para SynEdit, la pantalla es una grilla de celdas, en donde cada celda representa un carácter[1]:



Por otro lado, la información a mostrar en pantalla se almacena en una lista de cadenas, donde cada cadena representa a una línea.

Para manejar ambas representaciones, se usan dos tipos de coordenadas, en un SynEdit:

- **Coordenadas Físicas.** Referida a la posición en que aparece en pantalla un carácter, asumiendo que la pantalla se encuentra dividida en celdas de igual ancho.
- **Coordenadas Lógicas.** Referida a la posición del byte (o bytes) que representa al carácter, en la cadena.

Esta diferencia se hace notoria, sobre todo por el hecho de que el SynEdit maneja codificación UTF-8, lo que complica el manejo de las coordenadas en pantalla.

Las cadenas se almacenan como secuencias de bytes, pero lo que se muestra en pantalla son secuencias de caracteres ubicados en celdas.

La correspondencia de bytes a caracteres no es de 1 a 1[2].

- Un byte en la cadena puede representar a más de una carácter en pantalla[3]. Esto es cierto cuando se usan tabulaciones y están deben ser expandidas en varios espacios.

[1] Esto no es del todo cierto, porque algunos caracteres orientales pueden ocupar dos celdas del editor (full width).

[2] Inicialmente en los primeros editores de texto, siempre se mantenía una correspondencia de 1 a 1 (excepto cuando se soportaban tabulaciones), usando la codificación ASCCI o alguna similar.

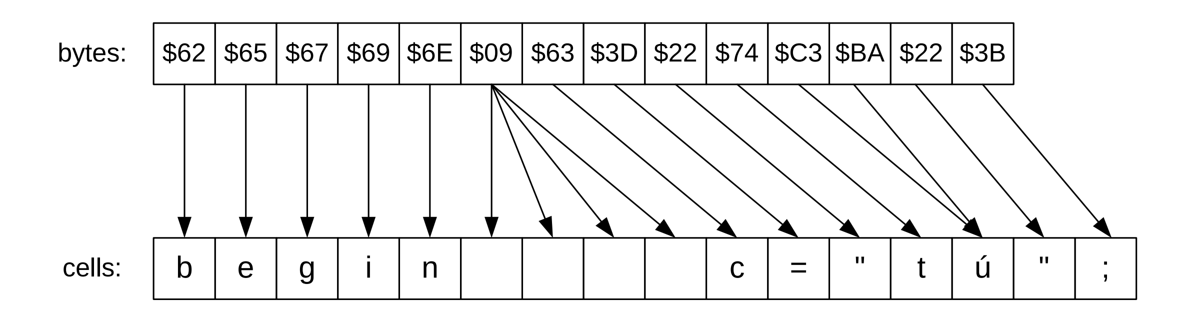
[3] Para complicar las cosas, si consideramos que un carácter en pantalla puede ocupar dos celdas del editor, en general, uno o más bytes de cadena pueden representar a una o dos celdas en la pantalla.

- Un carácter en pantalla puede ser representado por más de un byte en la cadena. Esto sucede porque SynEdit, maneja la codificación universal UTF-8, que en algunos casos (como las vocales acentuadas), asignan más de un byte por carácter.

El siguiente esquema muestra como se codifica una cadena de texto típica en SynEdit:

bytes	62	65	67	69	6E	09				63	3D	22	74	C3	BA	22	3B
celdas	b	e	g	i	n					c	=	"	t	ú		"	;

La tabulación se representa como 4 espacios, pero puede variar dependiendo de las configuraciones del editor. También se debe notar que en UTF-8, el carácter "ú" está representado por los bytes $C3 y $BA[4]. Otra forma de ver esta correspondencia, sería así:



Se puede apreciar, claramente, como las coordenadas lógicas son distintas a las coordenadas físicas. Aquí podemos encontrar que la coordenada lógica X de la letra "c" es 7, pero su coordenada física es 10.

Verticalmente la coordenada Y lógica y física son siempre iguales, así que no habrá que hacer transformaciones.

El cursor siempre trabaja en coordenadas físicas. Las coordenadas del cursor se encuentran en las propiedades CaretX, y CaretY.

CaretX va de 1 hasta el final de la línea. CaretY va de 1 hasta la cantidad de las líneas.

Por ejemplo para posicionar el cursor en el primer carácter de la segunda línea, haríamos:

```
SynEdit1.CaretX:=1;
SynEdit1.CaretY:=2;
```

También se puede hacer uso de la propiedad CaretXY, que es incluye las dos coordenadas X e Y en una estructura de tipo TPoint:

```
var Pos: TPoint;
  ...
  Pos.x:=5;
```

[4] Debido a la desigualdad en el tamaño de los caracteres en UTF-8, existen funciones específicas para el manejo de cadenas en UTF-8, como Utf8Length, and Utf8Pos.

```
  Pos.y:=2;
  SynEdit1.CaretXY := Pos;
```

En ciertas ocasiones puede resultar útil conocer las coordenadas del cursor en pixeles. En este caso se deben usar las propiedades:

```
SynEdit1.CaretXPix
SynEdit1.CaretYPix
```

La coordenada CaretXPix se mide desde el borde izquierdo del control (incluyendo el ancho del Panel Vertical o Gutter) y no desde el área editable del editor.

Para realizar las transformaciones entre las coordenadas lógicas y físicas, existen un grupo de funciones de transformación:

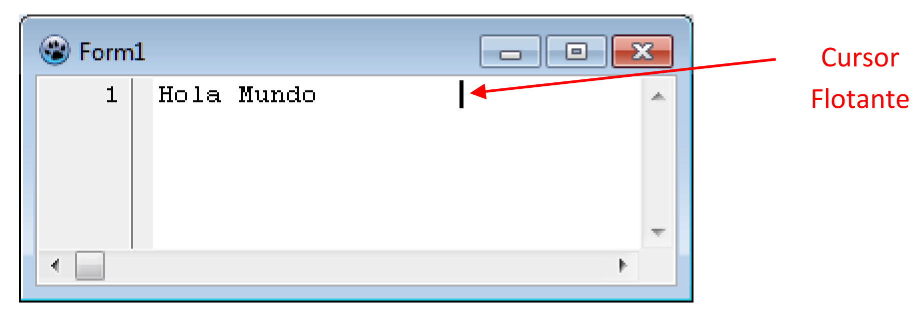
```
SynEdit1.LogicalToPhysicalCol();
SynEdit1.LogicalToPhysicalPos();
SynEdit1.PhysicalToLogicalCol();
SynEdit1.PhysicalToLogicalPos();
```

1.4.2 Manejo del cursor

Como ya vimos, la posición del cursor se fija con las propiedades CaretX y CaretY, pero existen ciertos límites con respecto a las ubicación del cursor.

En la práctica se puede decir que no hay límite práctico para el tamaño que puede tener una línea en SynEdit, a menos que se quiera exceder de 2 mil millones. Pero el cursor tiene más restricciones.

Un modo de trabajo extraño de SynEdit, es que puede ubicar al cursor más allá de los límites de la línea. Por ejemplo si la línea solo tiene 10 caracteres de ancho, el cursor podría ubicarse en la posición 20:



Esta ubicación extraña del cursor se puede lograr tanto por teclado como por medio del ratón.

A este efecto le llamo "Cursor Flotante" y es una característica poco usual en los editores de texto ya que lo normal es que no se pueda colocar el cursor más allá de los límites de la línea.

Este efecto solo es válido de forma horizontal ya que verticalmente el cursor estará siempre limitado por la cantidad de líneas que existan en el texto, o dicho de otro modo, la coordenada Y del cursor no puede ser mayor al número de líneas.

Para limitar la posición horizontal del cursor, dentro de los límites físicos de la línea actual, se debe usar esta configuración:

```
SynEdit1.Options := SynEdit1.Options + [eoKeepCaretX];
```

Sin embargo el modo de "cursor flotante", puede limitarse por medio de la propiedad "MaxLeftChar". Si se fija un valor máximo para "MaxLeftChar", no se permitirá ubicar horizontalmente más allá de este valor prefijado, a menos que exista una línea que exceda este tamaño. Es decir, que "MaxLeftChar" no limita el tamaño de la línea actual, pero puede limitar la posición horizontal del cursor.

Para forzar a ubicar el cursor sin considerar el tamaño de la línea destino, se puede usar el método MoveCaretIgnoreEOL(). Este método trabajará aun, cuando se exceda a "MaxLeftChar".

Otra peculiaridad, un tanto extraña de SynEdit, es que permite posicionar el cursor en medio de una tabulación como si se tratara de simples espacios, dando la impresión de que no existe una tabulación en esa posición.

Para notar la existencia de dicha tabulación, se puede intentar seleccionar un espacio dentro de la zona de tabulación. Si la selección no es posible, indicará que se está dentro de una tabulación.

Si se desea deshabilitar este comportamiento y forzar a tratar las tabulaciones como un solo carácter en el editor, se debe activar la opción "eoCaretSkipTab", de la propiedad Options2:

```
SynEdit1.Options2 := SynEdit1.Options2 + [eoCaretSkipTab];
```

La posición del cursor se puede cambiar a voluntad, usando las propiedades: CaretX, CaretY o CaretXY, como ya se ha mencionado, pero también se puede posicionar el cursor por comandos (Ver Sección 1.5.1 - Ejecutar comandos).

El siguiente código muestra como posicionar el cursor al final de todo el texto de SynEdit:

```
    SynEdit1.ExecuteCommand(ecEditorBottom, '',nil);
```

Para desplazar el cursor como si se usaran las teclas direccionales, se deben usar los comandos "ecLeft", "ecRight", "ecUp", y "ecDown".

1.4.3 Delimitador de línea

Como suele suceder en muchos aspectos de la informática, no hay un consenso en cuanto a cómo se define un salto de línea en un texto. En la actualidad se tienen 3 formas más conocidas de delimitar una línea:

1. Formato DOS: Caracteres #13#10. Usado en sistemas DOS/Windows.

2. Formato Unix: Caracter #10. Usado en sistemas Linux/Unix.
3. Formato MAC: Caracter #13. Usado en los sistemas MAC.

Cuando se lee un texto desde un archivo, usando SynEdit1.Lines.LoadFromFile(), se reconocerán cualquiera de estos delimitadores y se cargarán las líneas correctamente.

Sin embargo, si se desea grabar el contenido con SynEdit1.Lines.SaveToFile(), se puede perder la codificación original, dependiendo del sistema operativo usado o la configuración del sistema.

Para cambiar el salto de línea (DOS, UNIX, MAC) que se usa con SaveToFile(), se debe usar la unidad SynEditLines y ejecutar el método FileWriteLineEndType de la clase TSynEditLines:

```
Uses ..., SynEditLines;
...
If tipo = 'DOS' then TSynEditLines(SynEdi1.Lines).FileWriteLineEndType:=
sfleCrLf;
If tipo = 'UNIX' then  TSynEditLines(SynEdi1.Lines).FileWriteLineEndType:=
sfleLf;
If tipo = 'MAC' then  TSynEditLines(SynEdi1.Lines).FileWriteLineEndType:=
sfleCr;
```

Otra opción es cambiar la propiedad “TextLineBreakStyle” de la lista “SynEdit1.Lines”. Pero este separador solo tendrá efecto al usar la propiedad “Text ” de la lista.

1.5 Modificar el contenido

El contenido del editor SynEdit, usualmente será modificado por el usuario, usando el teclado, pero muchas veces requeriremos tomar el control del editor desde dentro del programa. Aquí describiremos las diversas formas de acceder al contenido del editor.

Para borrar todo el contenido del editor se debe usar el método “ClearAll”:

```
SynEdit1.ClearAll;
```

Para escribir en SynEdit, lo más fácil es asignando un texto a la propiedad Text:

```
SynEdit1.Text:='Hola mundo';
```

También podemos asignarle saltos de línea:

```
SynEdit1.Text:= 'Hola' + #13#10 + 'mundo';
```

Escribir de esta forma, hará que se pierda toda la información previa que pudiera contener el control (sin opción de deshacer), porque estamos asignándole un nuevo valor. Si solo quisiéramos agregarle información podríamos hacer:

```
SynEdit1.Text:=SynEdit1.Text + 'Hola mundo';
```

La propiedad Text, es una simple cadena y nos permite leer o escribir en el contenido del Editor. Como se trata de un capo de tipo cadena, se pueden realizar con él, las mismas operaciones que se hacen con las cadenas (búsqueda, concatenación, etc.).

Un detalle a tener en cuenta, es que cuando se lee el contenido de SynEdit, a través de su propiedad “Text”, se obtiene un salto de línea adicional, al final del texto.

Otra forma de insertar texto en el editor es usar el método InsertTextAtCaret():

```
SynEdit1.InsertTextAtCaret('texto insertado');
```

Con InsertTextAtCaret(), el texto se inserta directamente en la posición en donde se encuentra el cursor. Si hubiera alguna selección actual, se quita la selección (no se elimina el texto seleccionado) y se coloca el texto indicado en la posición del cursor.

El texto insertado puede ser de una o varias líneas.

Otra función útil para modificar el contenido de SynEdit es TextBetweenPoints(). El siguiente ejemplo muestra una forma rápida de remplazar un texto seleccionado con uno nuevo:

```
SynEdit1.TextBetweenPoints[ed.BlockBegin,ed.BlockEnd] := 'Nuevo texto';
```

Para información sobre BlockBegin y BlockEnd, ir a la sección 1.6 - Manejo de la selección.

1.5.1 Ejecutar comandos

El editor se puede controlar también, mediante el uso de comandos, con el método "ExecuteCommand".

Prácticamente todo lo que se puede hacer con el teclado en el editor, se puede hacer también con el uso de comandos. Por ejemplo, para insertar el caracter "x" en la posición actual del cursor, se debe usar:

```
SynEdit1.ExecuteCommand(ecChar, 'x',nil);
```

Existe un grupo enorme de comandos que se pueden ingresar a SynEdit. Todos ellos están declarados en la unidad "SynEditKeyCmds". Se muestran algunos de ellos:

```
  ecLeft            = 1;    // Move cursor left one char
  ecRight           = 2;    // Move cursor right one char
  ecUp              = 3;    // Move cursor up one line
  ecDown            = 4;    // Move cursor down one line

  ecDeleteLastChar  = 501;  // Delete last char (i.e. backspace key)
  ecDeleteChar      = 502;  // Delete char at cursor (i.e. delete key)
  ecDeleteWord      = 503;  // Delete from cursor to end of word
  ecDeleteLastWord  = 504;  // Delete from cursor to start of word
  ecDeleteBOL       = 505;  // Delete from cursor to beginning of line
  ecDeleteEOL       = 506;  // Delete from cursor to end of line
  ecDeleteLine      = 507;  // Delete current line
  ecClearAll        = 508;  // Delete everything
  ecLineBreak       = 509;   // Break line at current position, move caret to
new line
  ecInsertLine      = 510;  // Break line at current position, leave caret
  ecChar            = 511;  // Insert a character at current position
  ecSmartUnindent     = 512;   // NOT regocniced as command, used for group-
undo, set by beautifier

  ecImeStr          = 550;  // Insert character(s) from IME

  ecUndo            = 601;  // Perform undo if available
  ecRedo            = 602;  // Perform redo if available
  ecCut             = 603;  // Cut selection to clipboard
  ecPaste           = 604;  // Paste clipboard to current position

  ecBlockIndent     = 610;  // Indent selection
  ecBlockUnindent   = 611;  // Unindent selection
  ecTab             = 612;  // Tab key
  ecShiftTab        = 613;  // Shift+Tab key
  ...
```

Como se ven, los comandos pueden realizar todo tipo de acciones como pegar texto, borrar un carácter o deshacer los cambios.

Usando los comandos de ExecuteCommand(), se puede hacer todo tipo de modificaciones en SynEdit, pero es una forma lenta de modificación, porque se hace carácter por carácter.

Las acciones que se pueden realizar, incluyen también el manejo de los marcadores y el plegado de bloques de texto. Los siguientes comandos sirven para esta función:

```
  ecGotoMarker0      = 301;  // Goto marker
  ecGotoMarker1      = 302;  // Goto marker
  ecGotoMarker2      = 303;  // Goto marker
  ecGotoMarker3      = 304;  // Goto marker
  ecGotoMarker4      = 305;  // Goto marker
  ecGotoMarker5      = 306;  // Goto marker
  ecGotoMarker6      = 307;  // Goto marker
  ecGotoMarker7      = 308;  // Goto marker
  ecGotoMarker8      = 309;  // Goto marker
  ecGotoMarker9      = 310;  // Goto marker
  ecSetMarker0       = 351;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker1       = 352;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker2       = 353;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker3       = 354;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker4       = 355;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker5       = 356;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker6       = 357;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker7       = 358;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker8       = 359;  // Set marker, Data = PPoint - X, Y Pos
  ecSetMarker9       = 360;  // Set marker, Data = PPoint - X, Y Pos
  ecToggleMarker0     = 361;   // If marker is in the lie, remove marker, lese
set marker, Data = PPoint - X, Y Pos
  ecToggleMarker1    = 362;
  ecToggleMarker2    = 363;
  ecToggleMarker3    = 364;
  ecToggleMarker4    = 365;
  ecToggleMarker5    = 366;
  ecToggleMarker6    = 367;
  ecToggleMarker7    = 368;
  ecToggleMarker8    = 369;
  ecToggleMarker9    = 370;

  EcFoldLevel1      = 371;//fold all folds, greater/equal than nesting level 1
  EcFoldLevel2        = EcFoldLevel1 + 1;
  EcFoldLevel3        = EcFoldLevel2 + 1;
  EcFoldLevel4        = EcFoldLevel3 + 1;
  EcFoldLevel5        = EcFoldLevel4 + 1;
  EcFoldLevel6        = EcFoldLevel5 + 1;
  EcFoldLevel7        = EcFoldLevel6 + 1;
  EcFoldLevel8        = EcFoldLevel7 + 1;
  EcFoldLevel9        = EcFoldLevel8 + 1;
  EcFoldLevel0        = EcFoldLevel9 + 1;
```

```
  EcFoldCurrent       = 381;
  EcUnFoldCurrent     = 382;
  EcToggleMarkupWord = 383;
```

1.5.2 Accediendo a Lines[]

El contenido del editor (las líneas de texto) se almacena en la propiedad “Lines”, que es una lista de cadenas similar a un objeto “TStringList”, por lo tanto se puede acceder a el como a cualquier lista común de cadenas. Por ejemplo para mostrar el contenido de la primera línea, haríamos:

```
  showmessage(SynEdit1.Lines[0]);
```

Acceder a Lines[] es una forma rápida de acceso al contenido de SynEdit. Acceder de esta forma nos permite un manejo directo de cadenas (porque se trata de una lista de cadenas), por lo que podemos usar todas las funciones de cadena para nuestros propósitos. Empero, las modificaciones realizadas no podrán ser canceladas con el método “Undo” del editor.

Por ello no es recomendable modificar Lines[], si se pretende deshacer los cambios posteriormente.

Como Lines[], contiene todo el texto de SynEdit, si se quisiera acceder a la primera línea de SynEdit, debemos acceder a Lines[0]. Así para escribir un texto en la primera fila de SynEdit, debemos hacer:

```
      SynEdit1.Lines[0]:='Hola mundo';
```

Esta instrucción funcionará siempre, porque SynEdit contiene por lo menos una línea de texto, pero si intentáramos acceder a Lines[1], sin que existe una segunda línea en el editor, se generará un error en tiempo de ejecución.

Para conocer la cantidad de líneas del editor podemos usar el método Count:

```
      NumLineas := SynEdit1.Lines.Count;
```

Como Lines[] es una lista, comparte muchos de los métodos de las listas que conocemos. Por ejemplo para agregar una línea más al editor podemos hacer:

```
      SynEdit1.Lines.Add('Nueva línea de texto');
```

Algunas de las propiedades de Lines[] se listan en la siguiente tabla:

PROPIEDAD	DESCRIPCIÓN
Add	Agrega una línea de texto
AddStrings	Agrega todo el contenido de otra lista.
Capacity	Define la cantidad de líneas que se crean cuando se requiere ampliar la lista. Permite ahorrar tiempo ya que se crean varios elementos de una sola vez.

Clear	Limpia el contenido de la lista.
Count	Cantidad de elementos de Line[] (líneas del editor)
Delete	Elimina un elemento de la lista.
Exchange	Intercambia de posición dos líneas.
Insert	Inserta una línea en una posición específica.
LoadFromFile	Permite leer el contenido desde un archivo
SaveToFile	Permite grabar el contenido en un archivo
Text	Devuelve el contenido completo de todas las líneas del editor

Para iterar sobre todo el contenido de Lines[], se puede usar la siguiente construcción:

```
for i:= 0 to SynEdit1.Lines.Count-1 do
   ShowMessage(SynEdit1.Lines[i]);
```

Para realizar modificaciones, es preferible hacerlas usando comandos del editor que accediendo directamente a “Lines[]”, para mantener las opciones “Undo”, activas. Sin embargo, modificaciones complejas por comandos, pueden ser mucho más lentas que las modificaciones a Lines[].

1.5.3 El Portapapeles

Las funciones típicas con el portapapeles, se activan con estos métodos:

```
      SynEdit1.CopyToClipboard;
      SynEdit1.CutToClipboard;
      SynEdit1.PasteFromClipboard;
```

No es necesario identificar que hace, porque sus nombres son bastante conocidos, y ya sabemos que hacen un movimiento de datos entre el texto seleccionado y el portapapeles.

El comportamiento, es el mismo a como si ejecutáramos los atajos Ctrl+C, Ctrl+V, y Ctrl+X. De hecho esta misma combinación de teclas está activa por defecto, cuando se usa un SynEdit.

Es de resaltar que estas opciones, funcionarán aún en el modo de selección de columnas, de forma que se puedan copiar y pegar bloques rectangulares de texto.

Las opciones del portapapeles también se pueden acceder mediante el uso de comandos:

```
        SynEdit1.CommandProcessor(ecCopy, ' ', nil);
        SynEdit1.CommandProcessor(ecPaste, ' ', nil);
        SynEdit1.CommandProcessor(ecCut, ' ', nil);
```

Como el portapapeles trabaja con la selección, es usual trabajar con las propiedades “BlockBegin” y “BlockEnd”.

Es posible también colocar texto, directamente en el portapapeles, sin necesidad de hacer “Copy”:

```
SynEdit1.DoCopyToClipboard('texto a acopiar','');    //pone en portapapeles
```

1.5.4 Hacer y Deshacer.

SynEdit tiene un control muy bueno de los cambios realizados en el texto. Permite deshacer y rehacer los cambios en casi todos los casos. Estos cambios pueden ser inclusive modificaciones en modo columna.

Cada vez que se hace un cambio, SynEdit, guarda el cambio hecho en una memoria interna

Para controlar los cambios se usan los métodos:

MÉTODO/PROPIEDAD	DESCRIPCIÓN
SynEdit1.Undo()	Deshace un cambio hecho en pantalla
SynEdit1.Redo()	Rehace un cambio que ha sido desecho.
SynEdit1.ClearUndo()	Limpia la lista de cambios (Undo) y no permitirá deshacer a partir de ese punto.
SynEdit1.MaxUndo	Cantidad máxima de acciones (Undo) que se grabarán y podrán deshacerse.
SynEdit1.BeginUndoBlock() SynEdit1.EndUndoBlock()	Generan bloques únicos de cambios.
SynEdit1.CanUndo	Indica si hay acciones por deshacer
SynEdit1.CanRedo	Indica si hay acciones por rehacer

Casi todos los cambios hechos manualmente desde el teclado en un SynEdit, se pueden "deshacer", sin embargo, cuando se cambia el contenido desde código, se debe tener en cuenta que algunas acciones no podrán deshacerse.

La siguiente tabla muestra los métodos de cambio realizados en un SynEdit y si estos admiten "deshacer".

ACCIÓN	PERMITE DESHACER
Métodos: SynEdit1.ClearAll; SynEdit1.ExecuteCommand() SynEdit1.InsertTextAtCaret() SynEdit1.TextBetweenPoints() SynEdit1.SearchReplace() SynEdit1.SearchReplaceEx()	SÍ
Cambios de tipo: SynEdit1.Text:='Hola'; SynEdit1.LineText:='Hola';	NO
Modificaciones directas a SynEdit1.Lines[]	NO
Modificaciones usando el portapapeles: SynEdit1.CopyToClipboard; SynEdit1.CutToClipboard;	SÍ

SynEdit1.PasteFromClipboard;	

Hay que considerar los métodos que debemos usar para realizar los cambios en un SynEdit, si queremos mantener las opciones de “deshacer”.

Usualmente los cambios que generen, cada instrucción que modifique el contenido de SynEdit y que soporte “deshacer”, podrá ser desechada con una simple llamada a “Undo”.

Si se quisiera agrupar varias acciones para deshacerse con un solo “Undo”, se debe usar los métodos BeginUndoBlock y EndUndoBlock:

```
    SynEdit1.BeginUndoBlock;
    //Aquí pueden haber varios cambios que soporten deshacer.
    ...
    SynEdit1.EndUndoBlock;
```

Con esta construcción lograremos deshacer todos los cambios hechos con una sola llamada a “Undo”.

Puede ser práctico, usar la construcción BeginUpdate y EndUpdate, para evitar que se refresque el control hasta que se hayan terminado de realizar todos los cambios sobre SynEdit:

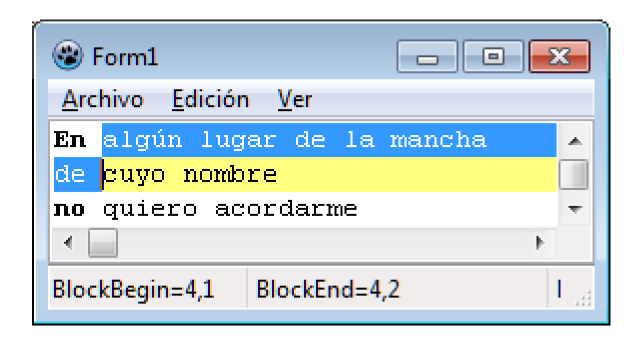
```
ActiveEditor.SynEditor.BeginUpdate;  //se deshabilita el refreso del SynEdit
try
    SynEdit1.BeginUndoBlock;
    //Aquí pueden haber varios cambios que soporten deshacer.
    ...
    SynEdit1.EndUndoBlock;
finally
  ActiveEditor.SynEditor.EndUpdate;  //Se reactiva el refresco de SynEdit
end;
```

Trabajar de esta forma nos permite mejorar la velocidad de los cambios, porque no se debe refrescar el control cada vez que se modifique algo.

Cuando esté deshabilitado el refresco de SynEdit, se ignorarán inclusive las peticiones de tipo Application.ProcessMessages().

1.6 Manejo de la selección

Como la mayoría de editores actuales, SynEdit maneja solo un bloque de selección[5]. Este bloque es el que se usa para las operaciones de cortado y copiado de texto, pero también se usa para modificar un texto por sobre-escritura:



La selección se define, por código, usando las propiedades "BlockBegin" y "BlockEnd" que son de tipo "Tpoint". En el ejemplo anterior, los valores asignados a "BlockBegin" y "BlockEnd", son:

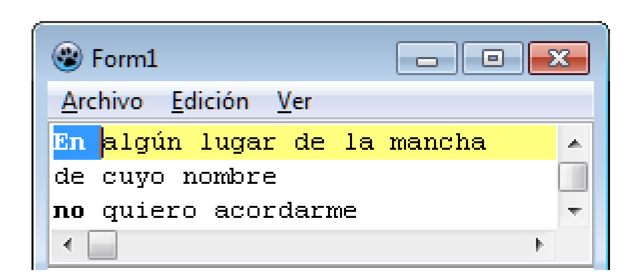
```
SynEdit1.BlockBegin.x:=4;
SynEdit1.BlockBegin.y:=1;
SynEdit1.BlockEnd.x:=4;
SynEdit1.BlockEnd.y:=2;
```

Cuando no hay selección activa, "BlockBegin" y "BlockEnd" indican la misma posición.

Otra forma de definir el bloque de selección es usar las propiedades "SelStart" y "SelEnd". Estas propiedades permiten trabajar de forma equivalente a "BlockBegin" y "BlockEnd", pero no son de tipo "TPoint", sino que son simples enteros que mapean el texto como si fueran una serie ininterrumpida de caracteres.

Los saltos de línea se consideran como dos caracteres en Windows y pueden también ser parte de la selección.

El siguiente ejemplo muestra una selección definida con SelStart = 1 y SelEnd = 4:



[5] Existen editores que pueden manejar más de un bloque de selección, pero la enorme mayoría solo trabaja con un solo bloque de selección.

El bloque de selección se define desde la posición del carácter apuntado por "SelStart", hasta el carácter anterior al carácter apuntado por "SelEnd". Por lo tanto, la cantidad de caracteres seleccionados será igual a (SelEnd-SelStart).

Cuando se desea seleccionar una región relativa a la posición actual del cursor, se pueden usar estos métodos:

SynEdit1.SelectWord;	Selecciona la palabra actual en donde se encuentra el cursor.
SynEdit1.SelectLine;	Selecciona la línea actual en donde se encuentra el cursor.
SynEdit1.SelectParagraph;	Selecciona el párrafo actual en donde se encuentra el cursor.
SynEdit1.SelectToBrace;	Selecciona el bloque delimitado por paréntesis, llaves o corchetes.

Todos estos métodos de selección se basan en la posición actual del cursor y funcionan como si se hiciera la selección de forma manual, considerando que SelectWord identifica una palabra considerando solo los caracteres alfabéticos, incluyendo los caracteres tildados y la letra ñ.

"SelectLine", tiene la siguiente declaración:

```
procedure SelectLine(WithLeadSpaces: Boolean = True);
```

EL parámetro opcional permite indicar si se quiere incluir los espacios en blanco inicial y final como parte de la selección. Si se pone en FALSE, la selección de la línea actual podría no ser completa si es que hubieran espacios iniciales o finales en la línea.

El método SelectToBrace, permite seleccionar bloques de texto que estén delimitados por paréntesis, llaves o corchetes. Solo funcionará cuando se cumplan las siguientes condiciones:

- Que el carácter actual donde se encuentre el cursor sea; '(', '{' o '['o que el carácter anterior al que se encuentre el cursor sea ')', '}', o ']'.
- Que existe el delimitador correspondiente en la misma línea o en otra línea cualquiera.

Además se debe considerar que la selección con SelectToBrace, permite el anidamiento de bloques del mismo tipo.

Para determinar si hay selección activa (texto seleccionado), se debe usar la propiedad "SelAvail":

```
if editor.SelAvail then    ...
```

Otra forma sería comparando las coordenadas de "BlockBegin" y "BlockEnd".

El texto seleccionado se puede obtener usando la propiedad "SelText". Solo se soporta un bloque de selección. Usualmente el cursor se encuentra en uno de los límites del bloque de selección, pero en modo de "bloque persistente", se puede hacer al cursor independiente de la posición del bloque de selección.

La propiedad “SelText”, es también de escritura, de modo que nos permite modificar el texto seleccionado. El siguiente código elimina el texto seleccionado:

```
SynEdit1.SelText := '';
```

Sin embargo, para borrar la selección existe el método “ClearSelection”, que es una forma abreviada.

Para seleccionar todo el texto, debemos usar:

```
SynEdit1.SelectAll;
```

Para eliminar el texto seleccionado se puede usar el método “ClearSelection”:

```
SynEdit1.ClearSelection;
```

También se puede usar enviar el comando “ecDeleteLastChar “ con ExecuteCommand():

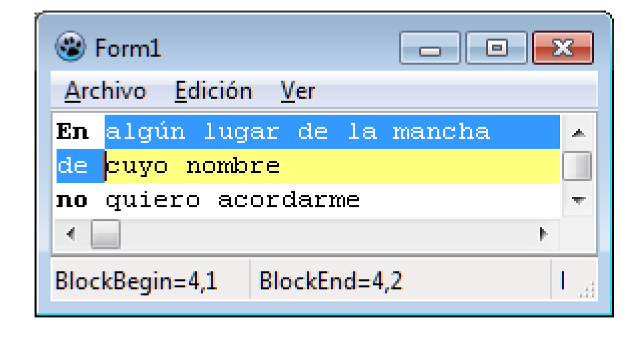
```
SynEdit1.ExecuteCommand(ecDeleteLastChar,'',nil);
```

Otra forma alternativa, sería remplazar el texto seleccionado con TextBetweenPoints():

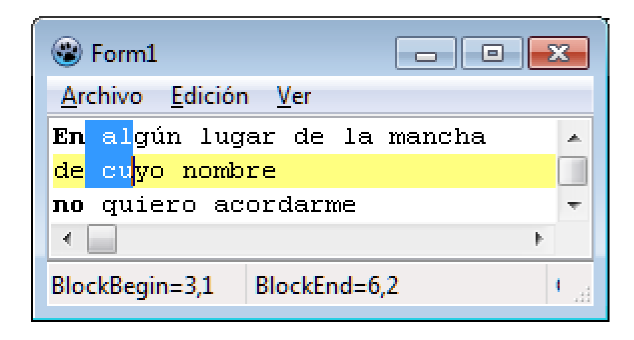
```
SynEdit1.TextBetweenPoints[editor.BlockBegin,editor.BlockEnd] := '';
```

1.6.1 Selección en Modo Columna

La selección de texto en SynEdit, se puede hacer de diversas formas. Lo normal es que se seleccionen todas las filas entre el inicio y el fin del bloque de selección:



Pero también se puede usar el modo de selección en columnas:



En este modo el área de selección forma un rectángulo e incluye solo parcialmente a las líneas en su camino.

Para seleccionar en modo columna en SynEdit, se debe usar la combinación de teclas <Alt>+<Shift>+direccionales.

Una vez que se tiene el texto seleccionado se pueden ejecutar las acciones de cortado, copiado o pegado. También se puede sobrescribir la selección pulsando cualquier tecla.

Hay que notar que cualquier tecla pulsada que no sea la combinación, <Alt>+<Shift>+direccionales, hará que el modo de selección en modo columna termine.

Para pasar al modo columna por programa, se puede usar el siguiente código:

```
uses ... , SynEditTypes;

var pos:Tpoint;
  ...
  pos.x:=3;
  pos.y:=2;
  SynEdit1.BlockBegin:= pos;   //define punto inicial de selección
  pos.x:=8;
  pos.y:=3;
  SynEdit1.BlockEnd:= pos;     //define punto final de selección
  SynEdit1.CaretXY := pos;
  SynEdit1.SelectionMode:=smColumn;  //cambia a modo columna
  ...
```

De igual forma, en este caso, cualquier tecla pulsada que no sea la combinación, <Alt>+<Shift>+direccionales, hará que el modo de selección en modo columna termine.

Existen otras formas de selección, definidas en la unidad “SynEditTypes”:

- smNormal,
- smColumn
- smLine,

El modo smNormal, es el modo que está activo por defecto y es el modo de selección normal.

El modo smColumn, es el modo de selección por columnas.

El modo smLine, es un modo de selección que hará que todas las líneas entre BlockBegin y BlockEnd, se marquen como parte de la selección.

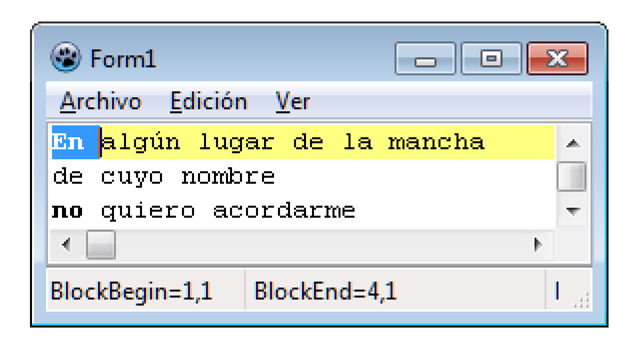
También se puede realizar selección por columnas usando los comandos de teclado:

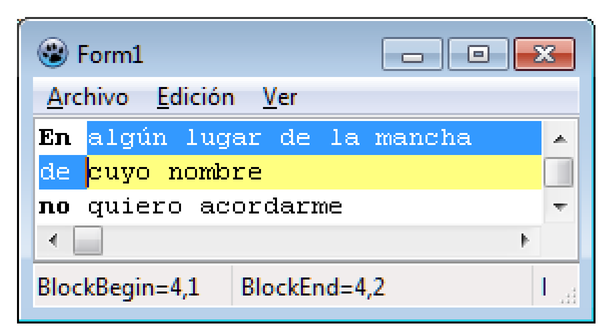
```
SynEdit1.ExecuteCommand(ecColSelUp, #0, nil)
```

Las constantes ecColSelUp, ecColSelDown, ecColSelLeft, ecColSelRight, y otras más, permiten desplazar el cursor manteniendo el modo columna, que es equivalente a mantener pulsadas las teclas <Alt>+<Shift>.

1.6.2 BlockBegin y BlockEnd

Determinan las coordenadas del bloque de selección.





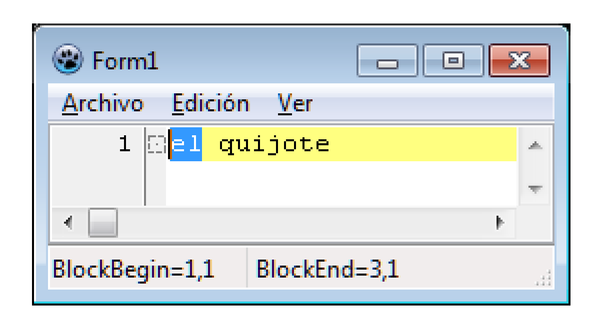
Se pueden definir bloques de selección usando BlockBegin y BlockEnd. Por ejemplo si se tiene un editor "ed" de tipo TSynEdit, se puede seleccionar con este código:

```
  p: Tpoint;
...
  ed.Text:='el quijote';

  //fija inicio de bloque
  p:= ed.BlockBegin;
  p.x:=1;
  ed.BlockBegin := p;
```

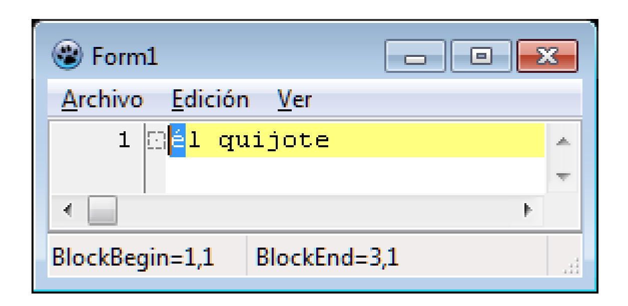
```
//fija fin de bloque
p:= ed.BlockEnd;
p.x:=3;
ed.BlockEnd := p;
```

El código generará la siguiente salida en pantalla:



Sin embargo, si el SynEdit trabajara en UTF-8 (como suele hacerlo), un carácter especial, puede tener dos caracteres de ancho. Por ello debe tenerse cuidado con las coordenadas lógicas (las que maneja BlockBegin y BlockEnd) y las coordenadas físicas.

Si la cadena en el editor hubiera sido "él quijote", la selección tendría otro resultado:

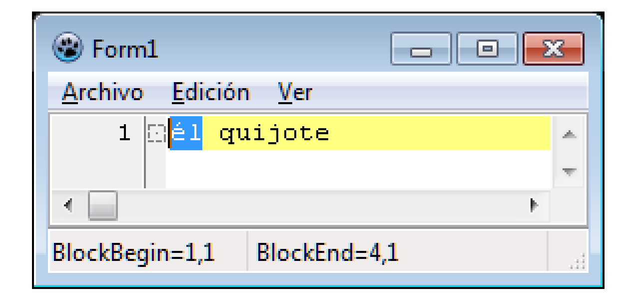


Para salvar este comportamiento, se debe usar la función PhysicalToLogicalCol():

```
ed.Text:='él quijote';
//fija inicio de bloque
p:= ed.BlockBegin;
p.x:=1;
ed.BlockBegin := p;

//fija fin de bloque
p:= ed.BlockEnd;
p.x:= ed.PhysicalToLogicalCol(ed.Lines[0],0, 3);
ed.BlockEnd := p;
```

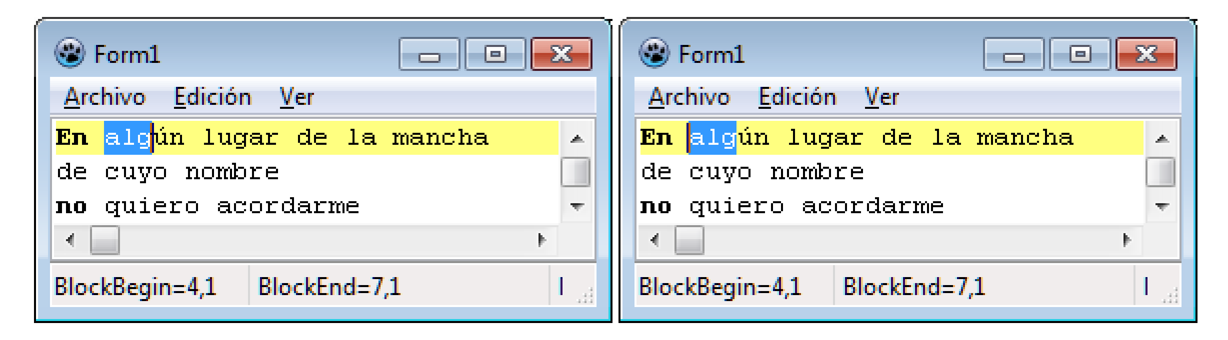
Ahora el comportamiento, es el esperado:


Form1
Archivo Edición Ver
1
él quijote
BlockBegin=1,1
BlockEnd=4,1

1.6.3 BlockBegin y BlockEnd en Selección Normal

Si la selección solo tiene una fila, el punto BlockBegin apuntará siempre en la columna de la izquierda de la selección, no importa el sentido desde donde se haya hecho la selección.

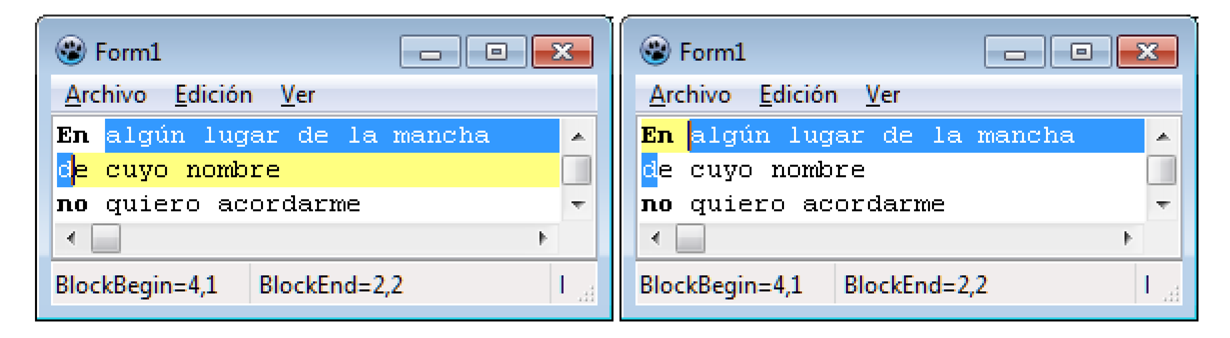
En los siguientes ejemplos, se ha hecho una selección de izquierda a derecha y de derecha a izquierda respectivamente.



Cómo se ve, no hay diferencia en el sentido de la selección.

Si la selección tiene varias filas, el punto BlockBegin siempre aparecerá en la fila superior, no importa el sentido desde donde se haya hecho la selección.

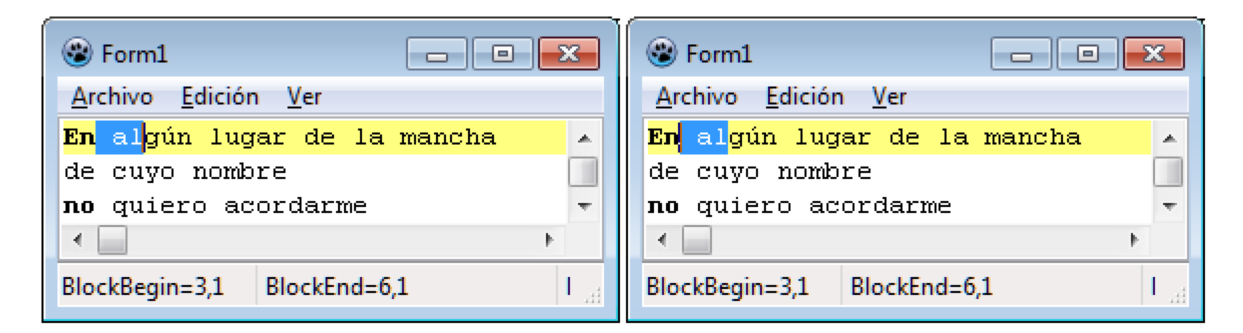
En los siguientes ejemplos, se ha hecho una selección de izquierda a derecha y de derecha a izquierda respectivamente.



1.6.4 BlockBegin y BlockEnd en Selección en Modo Columna

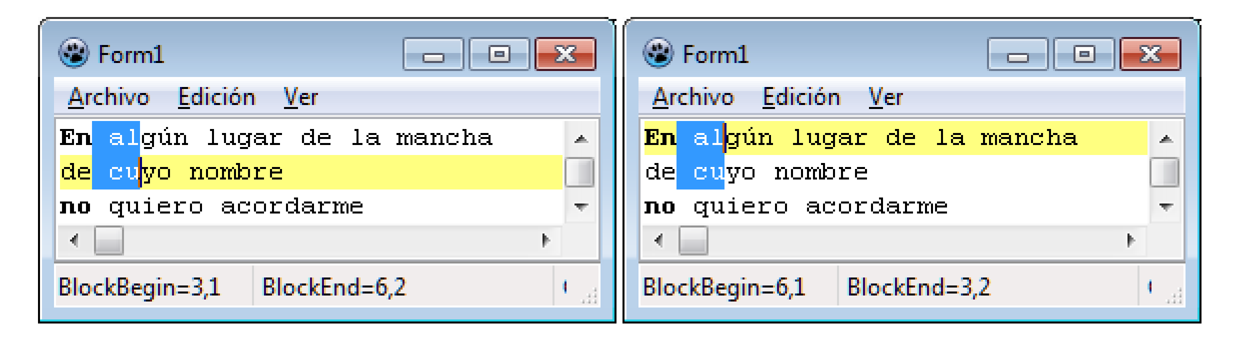
Si la selección solo tiene una fila, el punto BlockBegin apuntará siempre en la columna de la izquierda de la selección, no importa el sentido desde donde se haya hecho la selección.

En los siguientes ejemplos, se ha hecho una selección de izquierda a derecha y de derecha a izquierda respectivamente.



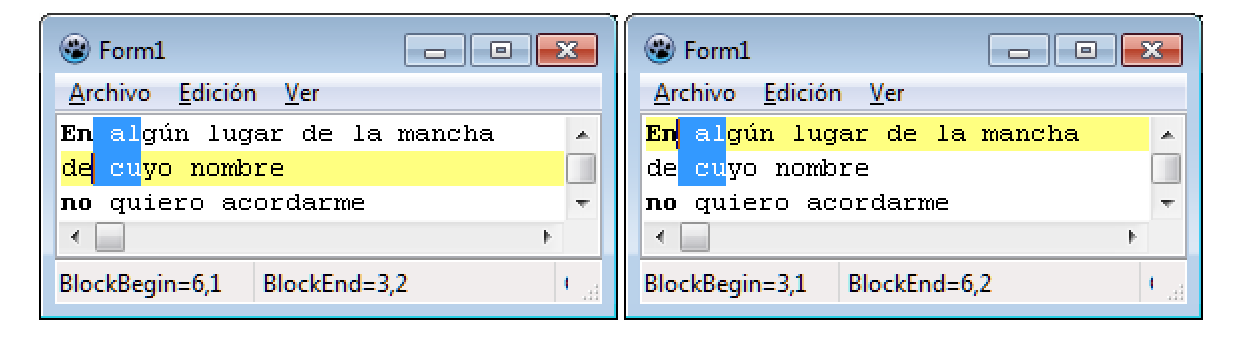
Si la selección tiene varias filas, el punto BlockBegin siempre aparecerá en la fila superior, no importa el sentido desde donde se haya hecho la selección.

En los siguientes ejemplos, se ha hecho una selección de izquierda a derecha, hacia abajo y hacia arriba:



Observar que en el último caso, BlockBegin, toma la ubicación del cursor.

En los siguientes ejemplos, se ha hecho una selección de derecha a izquierda, hacia abajo y hacia arriba:



Las posiciones de BlockBegin y BlockEnd son inversas al caso anterior.

En todos los casos vistos el cursor siempre tomará el valor de BlockBegin o de BlockEnd.

1.7 Búsqueda y Reemplazo

Casi todo editor de texto, tiene incluido opciones para búsqueda y reemplazo. Así que es útil saber cómo implementar las opciones de búsqueda y reemplazo en SynEdit.

Después de conocer algunas de sus propiedades y métodos, podríamos nosotros mismos implementar las funcione de búsqueda y reemplazo. Para la búsqueda podríamos explorar línea a línea, el contenido de SynEdit (accediendo a la lista Lines[]) y seleccionar el texto buscado, usando las propiedades BlockBegin y BlockEnd.

Sin embargo SynEdit, tiene ya incluidas, dos funciones para la búsqueda reemplazo:

```
function SearchReplace(const ASearch, AReplace: string;
      AOptions: TSynSearchOptions): integer;

function SearchReplaceEx(const ASearch, AReplace: string;
      AOptions: TSynSearchOptions; AStart: TPoint): integer;
```

SearchReplaceEx() es similar a SearchReplace(), con la diferencia de que SearchReplaceEx(), busca desde la posición indicada (AStart).

Estas funciones nos proveen de las funcionalidades de búsqueda hacia arriba o abajo, con sensibilidad de caja (mayúscula/minúscula), palabra completa, o el uso de expresiones regulares.

El valor devuelto por estas funciones, es un número entero:

MODO	VALOR	SIGNIFICADO
Búsqueda	0	Elemento no encontrado
Búsqueda	1	Se encontró al menos un elemento.
Reemplazo	0	Elemento no encontrado
Reemplazo	n	Se reemplazó “n” elementos

En modo Búsqueda, estas funciones se detienen al encontrar la primera coincidencia, y al seleccionan, además de hacer el texto seleccionado visible en el editor.

El parámetro AOptions es un conjunto que puede incluir los siguientes elementos:

```
  TSynSearchOption =
    ( ssoMatchCase,
      ssoWholeWord,
      ssoBackwards,
      ssoEntireScope,
      ssoSelectedOnly,
      ssoReplace,
      ssoReplaceAll,
      ssoPrompt,
      ssoSearchInReplacement,   //continue search-replace in replacement
                                //(with ssoReplaceAll) replace recursive
      ssoRegExpr,
      ssoRegExprMultiLine,
      ssoFindContinue    // Assume the current selection is the last match,
```

```
                            // and start search behind selection (before
                            // if ssoBackward)
                            // Default is to start at caret (Only SearchReplace /
                            // SearchReplaceEx has start/end param)
    );
```

Todas estas constantes están definidas en la unidad “SynEditTypes”.

Cuando se incluyen los elementos ssoReplace o ssoReplaceAll, se hará un remplazo, de otra forma se hará solamente una búsqueda.

1.7.1 Búsqueda

Para buscar una cadena simple, se puede usar el siguiente código:

```
var
  encon : integer;
  buscado : string;
begin
  buscado := 'texto a buscar';
  encon := editor.SearchReplace(buscado,'',[]);
  if encon = 0 then
     ShowMessage('No se encuentra: ' + buscado);
...
```

Cuando no se especifican opciones de búsqueda, se asumen las siguientes opciones;

- La búsqueda en SynEdit, se iniciará siempre desde la posición del cursor.
- La dirección de búsqueda, es hacia adelante del cursor hasta el final del archivo.
- No hay sensibilidad por caja, se identifican por igual mayúsculas o minúsculas.

Al llamar a SearchReplace() se produce el siguiente flujo:

1. Inicia la búsqueda con el texto solicitado y las opciones indicadas.
2. Si la búsqueda no tuvo éxito, se devuelve cero y se sale de la función.
3. Si la búsqueda tuvo éxito, se selecciona la primera coincidencia, y se hace el texto seleccionado visible en el editor.

El hecho de seleccionar el texto encontrado, ocasiona que el cursor se traslade al final del texto seleccionado. De modo que la siguiente llamada a SearchReplace(), buscará a partir de esta posición (buscar siguiente).

Para realizar una búsqueda hacia atrás:

```
  encon := editor.SearchReplace(buscado,'',[ssoBackwards]);
```

Para realizar una búsqueda teniendo en cuenta la caja:

```
  encon := ed.SearchReplace(buscado,'',[ssoMatchCase]);
```

Las opciones de búsqueda se pueden combinar.

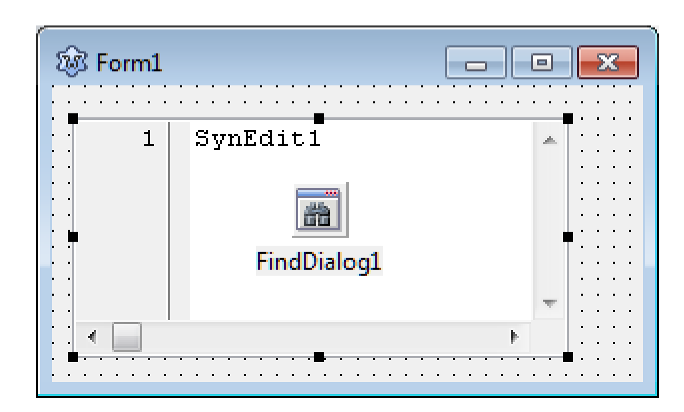
Cuando se indica la opción “ssoEntireScope”, la búsqueda se hace siempre desde el principio del texto (independientemente de donde se encuentre el cursor) y se detiene al encontrar el primer elemento.

Cuando se indica la opción “`ssoSelectedOnly`”, la búsqueda se hará dentro del texto seleccionado, deteniéndose al encontrar el primer elemento. Si se intenta una realizar una siguiente búsqueda, se dará siempre el mismo resultado porque el texto seleccionado, después de llamar a SearchReplace(), contendrá siempre el texto buscado.

1.7.2 Búsqueda usando TFindDialog

Existe entre los controles de Lazarus, un diálogo creado específicamente para operaciones de búsqueda. Este componente es FIndDialogo, y se encuentra en la paleta de componentes, en la pestaña “Dialogs”.

Para usarlo, debemos colocar este componente en nuestro formulario:



Y luego debemos crear un procedimiento para atender al evento “OnFind”.

El procedimiento del evento puede tener la siguiente forma:

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  encon   : integer;
  buscado : string;
  opciones: TSynSearchOptions;
begin
  buscado := FindDialog1.FindText;
  opciones := [];
  if not(frDown in FindDialog1.Options) then opciones += [ssoBackwards];
  if frMatchCase in FindDialog1.Options then opciones += [ssoMatchCase];
  if frWholeWord in FindDialog1.Options then opciones += [ssoWholeWord];
  if frEntireScope in FindDialog1.Options then opciones += [ssoEntireScope];

  encon := editor.SearchReplace(buscado,'',opciones);
```

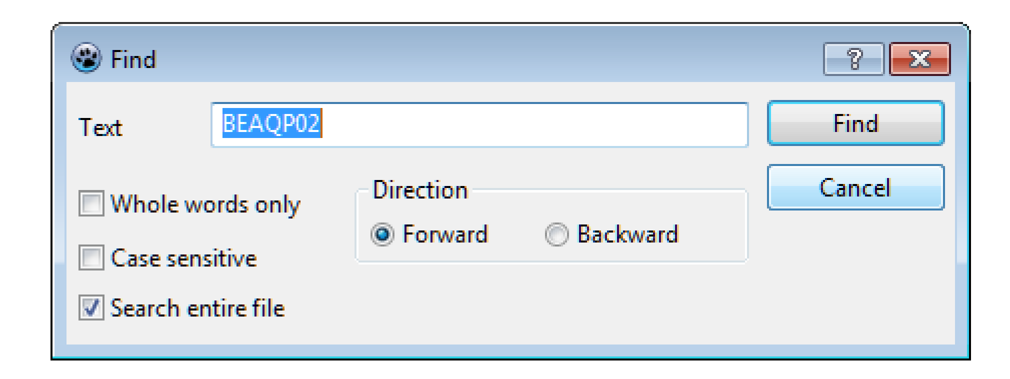
```
  if encon = 0 then
     ShowMessage('No se encuentra: ' + buscado);
end;
```

El control “FindDialog1”, expone las opciones seleccionadas a través de su propiedad “Options”.

La idea es pasarle las opciones seleccionadas en el diálogo a la variable “opciones” antes de llamar a SearchReplace().

Ahora desde alguna parte estratégica de nuestro programa (como la respuesta al menú), debemos incluir el código para abrir el diálogo y poder iniciar la búsqueda:

```
procedure TForm1.acBuscarExecute(Sender: TObject);  //Búsqueda
begin
  FindDialog1.Execute;  //abre el diálogo de búsqueda
end;
```



Al pulsar sobre “Find”, se llamará al evento OnFind, que debe estar asociado al método “`FindDialog1Find`” que hemos creado. El diálogo, sin embargo permanecerá visible, hasta pulsar en “Cancel” o pulsar la tecla <escape>.

El método “`FindDialog1Find`”, tiene también otra utilidad. Sirve para implementar la funcionalidad “Buscar siguiente”, porque se puede llamar a este método, aun cuando el diálogo “`FindDialog1`” se encuentre oculto.

Si esta ventana no se adapta a las necesidades de búsqueda, requeridas se puede crear siempre un formulario especial para nuestra búsqueda personalizada.

1.7.3 Reemplazo

El proceso de reemplazo es similar al de búsqueda. Para buscar una cadena simple, se puede usar el siguiente código:

```
var
  encon : integer;
  buscado : string;
begin
  buscado := FindDialog1.FindText;
```

```
  encon := editor.SearchReplace(buscado,'nueva cadena',[ssoReplace]);
  if encon = 0 then
     ShowMessage('No se encuentra: ' + buscado);
...
```

La diferencia está en que se debe indicar la opción “ssoReplace” al método SearchReplace(), además de indicar el texto de reemplazo a usar.

El modo de funcionamiento de SearchReplace(), en modo de reemplazo es similar al modo de búsqueda:

1. Inicia la búsqueda con el texto solicitado y las opciones indicadas.
2. Si la búsqueda no tuvo éxito, se devuelve cero y se sale de la función.
3. Si la búsqueda tuvo éxito, se reemplaza la primera coincidencia, y se hace el texto reemplazado visible en el editor.
4. Se deja el cursor al final del texto reemplazado, listo para otra secuencia de búsqueda.

Este modo de funcionamiento es útil en modo de búsqueda, pero en modo de reemplazo puede resultar extraño, ya que no se pide ninguna confirmación para remplazar el texto y el reemplazo se produce instantáneamente, sin ninguna selección previa.

Para mejorar este comportamiento, se puede agregar una ventana de confirmación, antes del reemplazo, de modo que sirva, a la vez, para ver el texto que va a ser reemplazado.

No hay un diálogo predefinido para crear una ventana de confirmación. Si queremos usar una tendremos que crearlo nosotros mismos.

Un diálogo sencillo que nos podría servir sería un `MessageBox()` con los botones Si-No-Cancelar. De modo que nuestro procedimiento de reemplazo, podría escribirse de la siguiente forma:

```
var
  encon, r: integer;
  buscado : string;
  opciones: TSynSearchOptions;
  ed      : TSynEdit;
begin
  ...
  buscado := 'cadena buscada';
  opciones := [];   //opciones de búsqueda
  encon := ed.SearchReplace(buscado,'',opciones);  //búsqueda
  while encon <> 0 do begin
      //pregunta
      r          :=          Application.MessageBox('¿Reemplazar          esta
ocurrencia?','Reemplazo',MB_YESNOCANCEL);
      if r = IDCANCEL then exit;
      if r = IDYES then begin
        ed.TextBetweenPoints[ed.BlockBegin,ed.BlockEnd] := 'nueva cadena';
      end;
```

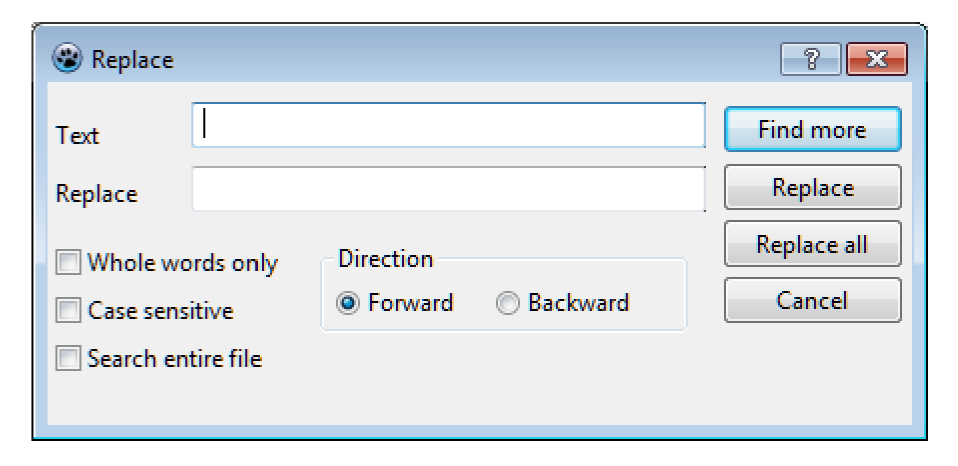
```
      //busca siguiente
      encon := ed.SearchReplace(buscado,'',opciones);  //búsca siguiente
  end;
  ShowMessage('No se encuentra: ' + buscado);
end.
```

La idea aquí es preguntar antes de cada reemplazo, daño la opción de omitir alguna ocurrencia o cancelar todo el proceso.

En este modo de trabajo, no estamos haciendo uso de la opción “ssoReplace”, sino que usamos SearchReplace() , únicamente en modo de búsqueda. El reemplazo lo hacemos en el editor usando el método “TextBetweenPoints()”.

1.7.4 Reemplazo usando TReplaceDialog

Así como para la búsqueda existe el diálogo TFindDialog, también es posible usar el diálogo “TReplaceDialog” desde la paleta de componentes.



Este diálogo nos facilita la entrada de datos para iniciar una Búsqueda/Reemplazo, pero tampoco nos proporciona un diálogo de confirmación para antes de remplazar.

El modo de trabajo con este diálogo es sencillo. Se deben asignar los eventos “OnFind” y “OnReplace”. El primer evento se ejecutará cuando se pulse el botón “Find more” y el segundo se ejecutará con el botón “Replace” o con el botón “Replace all”.

El siguiente código, es un ejemplo de como implementar la opción de reemplazo usando el diálogo “TReplaceDialog”:

```
procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
var
  encon, r : integer;
  buscado : string;
  opciones: TSynSearchOptions;
begin
  buscado := ReplaceDialog1.FindText;
  opciones := [ssoFindContinue];
  if not(frDown in ReplaceDialog1.Options) then opciones += [ssoBackwards];
  if frMatchCase in ReplaceDialog1.Options then opciones += [ssoMatchCase];
```

```
  if frWholeWord in ReplaceDialog1.Options then opciones += [ssoWholeWord];
  if    frEntireScope    in    ReplaceDialog1.Options    then    opciones    +=
[ssoEntireScope];
  if frReplaceAll in ReplaceDialog1.Options then begin
    //se ha pedido reemplazar todo
    encon := ed.SearchReplace(buscado,ReplaceDialog1.ReplaceText,
                              opciones+[ssoReplaceAll]);  //reemplaza
    ShowMessage('Se reemplazaron ' + IntToStr(encon) + ' ocurrencias.');
    exit;
  end;
  //reemplazo con confirmación
  ReplaceDialog1.CloseDialog;
  encon := ed.SearchReplace(buscado,'',opciones);  //búsqueda
  while encon <> 0 do begin
      //pregunta
      r          :=          Application.MessageBox('¿Reemplazar          esta
ocurrencia?','Reemplazo',MB_YESNOCANCEL);
      if r = IDCANCEL then exit;
      if r = IDYES then begin
        ed.TextBetweenPoints[ed.BlockBegin,ed.BlockEnd]                      :=
ReplaceDialog1.ReplaceText;
      end;
      //busca siguiente
      encon := ed.SearchReplace(buscado,'',opciones);  //búsca siguiente
  end;
  ShowMessage('No se encuentra: ' + buscado);
end;
```

Este código corresponde al que debe asociarse al evento “OnReplace”. El código para el evento “OnFind”, puede ser una simple búsqueda:

```
procedure TForm1.ReplaceDialog1Find(Sender: TObject);
var
  encon   : integer;
  buscado : string;
  opciones: TSynSearchOptions;
begin
  buscado := ReplaceDialog1.FindText;
  opciones := [];
  if not(frDown in ReplaceDialog1.Options) then opciones += [ssoBackwards];
  if frMatchCase in ReplaceDialog1.Options then opciones += [ssoMatchCase];
  if frWholeWord in ReplaceDialog1.Options then opciones += [ssoWholeWord];
  if    frEntireScope    in    ReplaceDialog1.Options    then    opciones    +=
[ssoEntireScope];

  encon := ed.SearchReplace(buscado,'',opciones);
  if encon = 0 then
     ShowMessage('No se encuentra: ' + buscado);
end;
```

En ambos ejemplos, se asume que se ha incluido en el formulario el editor “ed”, y el diálogo ”ReplaceDialog1”.

Este diálogo, tiene diversas opciones que pueden ser personalizadas desde la propiedad “Options”. Estas opciones permiten ocultar o mostrar ciertos botones.

La opción “frPromptOnReplace” permite pasar el control a un formulario de confirmación, así como hicimos con `MessageBox()`, pero esta confirmación se pide a través del evento “OnReplaceText” del editor de texto (no del diálogo), que debe tener la forma:

```
TReplaceTextEvent = procedure(Sender: TObject;
                     const ASearch, AReplace: string;
                     Line, Column: integer;
                     var ReplaceAction: TSynReplaceAction) of object;
```

1.8 Opciones de remarcado

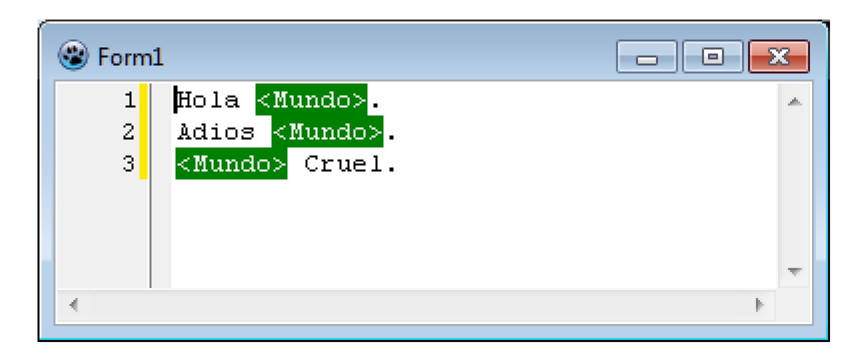
SynEdit es un componente bastante completo. Entre sus diversas opciones, incluye remarcado del contenido. Estas opciones están incluidas en el mismo código del editor y son independientes del uso de los resaltadores de sintaxis, descritos en la sección 2.

1.8.1 Remarcado de un texto.

Para marcar un texto cualquiera, se puede usar el método SetHighlightSearch(), de la siguiente forma:

```
uses ..., SynEditTypes;
...
SynEdit1.HighlightAllColor.Background := clGreen;
SynEdit1.SetHighlightSearch('<Mundo>', [ssoSelectedOnly]) ;
```

Si se aplica este código a un texto que contenga la secuencia de caracteres <Mundo>, se obtendrá un resultado como el mostrado:



Hay que indicar que el texto a resaltar, puede ser cualquier combinación de caracteres, no necesariamente letras.

Para especificar que solo se marquen palabras completas, se puede usar la opción "ssoWholeWord":

```
  SynEdit1.HighlightAllColor.Background := clGreen;
  SynEdit1.SetHighlightSearch('Mundo', [ssoSelectedOnly,ssoWholeWord]);
```

Este tipo de remarcado, utiliza las mismas opciones que cuando se realiza una búsqueda de texto (Ver Sección 1.7 - Búsqueda y Reemplazo), así que tiene las mismas consideraciones.

El texto puede modificarse, pero cada vez que se encuentre la secuencia buscada, se marcará nuevamente el texto. Es como tener implementado un buscador permanente de texto.

Este método de remarcado de texto, funciona como un resaltador sencillo de sintaxis, pero es limitado porque solo se aplica a un solo texto y porque solo reconoce identificadores y símbolos.

1.8.2 Remarcado de la palabra actual

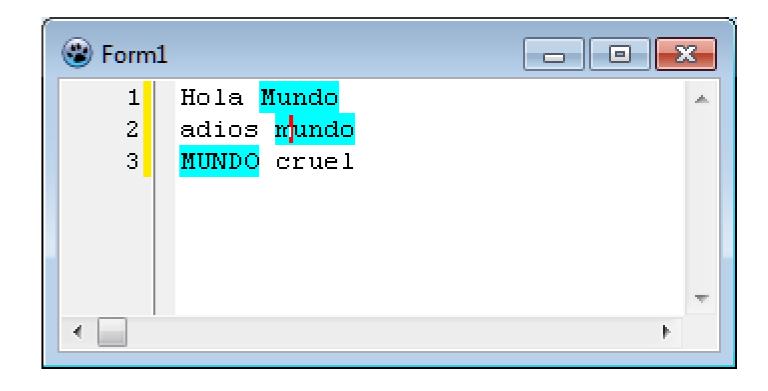
Existe otra forma de remarcado de texto que es aplicable solamente a palabras y en la posición actual del cursor. Una palabra solo puede contener caracteres alfanuméricos, el signo dólar y el guion bajo. También se incluyen las vocales acentuadas y la letra ñ.

Para hacer uso de esta característica, necesitamos incluir la unidad "SynEditMarkupHighAll", y crear un objeto de la clase "TSynEditMarkupHighlightAllCaret".

```
uses ... , SynEditMarkupHighAll;
...
var
  SynMarkup: TSynEditMarkupHighlightAllCaret;
begin
  //Inicia resaltado de palabras iguales
  SynMarkup := TSynEditMarkupHighlightAllCaret(
                    SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret]);
  SynMarkup.MarkupInfo.Background := clAqua;

  SynMarkup.WaitTime := 250; // tiempo en millisegundos
  SynMarkup.Trim := True;
  SynMarkup.FullWord:= True; //solo marcará coincidencia en palabra completa
```

Este código hará que se marquen las palabras iguales a la palabra en la que está el cursor, tal como se muestra en la siguiente figura:

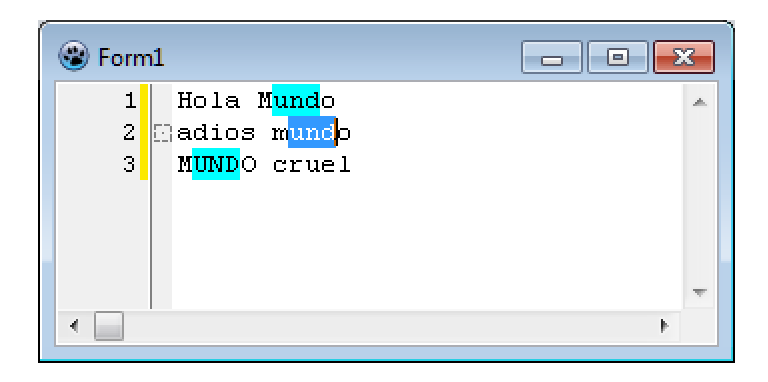


El editor auotmáticamente reconoce la palabra que está debajo del cursor y toma esta para realizar el marcado en todo el texto. Por defecto se ignora el tipo de caja (mayúscula o minúscula).

El remarcado, puede incluir diversos atributos como color de fondo, color de texto, o borde. Estos atributos se acceden mediante la propiedad "MarkupInfo".

El valor fijado en "WaitTime" indicará cuantos milisegundos se esperará para reconocer la palabra en donde se encuentra el cursor. SI no se especifica, se esperará 2 segundos. No es recomendabe usar un valor muy bajo para "WaitTime", porque se podría estar forzando al editor a realizar búsquedas muy frecuentes.

Esta opción de remarcado, funciona también con el texto seleccionado. De forma que se buscará solo el texto que se encuentre dentro de la selección:



Hay que notar que el remarcado no lee todo el contendio del editor completo. Por defecto solo leerá hasta 100 líneas adelnate y atrás de la pantalla actual.

Para desactivar el remarcado de la palabra actual, se puede deshabilitar el marcador asignado a SynEdit:

```
SynMarkup.Enabled:= False; //desactiva el resaltado
```

1.8.3 Remarcado de la línea actual

Muchas veces resulta conveniente marcar la línea actual para identificar fácilmente, dónde se encuentra el cursor. SynEdit permite cambiar fácilmente el color de fondo de la línea en donde se encuentra el cursor:

```
SynEdit1.LineHighlightColor.Background:=clYellow;
```

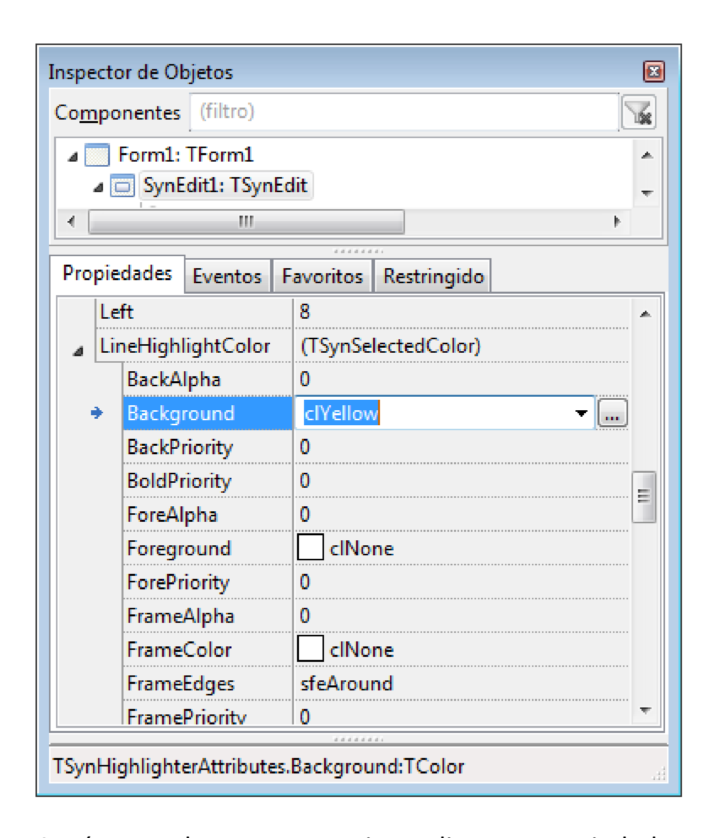
La instrucción anterior pintará de amarillo, el fondo de la línea actual. El resultado será similar al mostrado en la siguiente figura:

```
Form1
Este componente SynEdit1 está
configurado para resaltar la
línea actual, es decir, la línea
en la que se encuentra el cursor.
```

Para desactivar el resaltado de la línea actual, se puede usar el mismo color de fondo del editor, o se puede usar el color clNone:

```
SynEdit1.LineHighlightColor.Background:=clNone;
```

El resaltado de la línea actual, se puede activar también, usando el inspector de objetos, configurando la propiedad LineHighlightColor:



Aquí se puede notar que existen diversas propiedades adicionales a "Background", para cambiar la apariencia que tendrá la línea resaltada. Todas ellas se pueden configurar por código, o desde el inspector de objetos.

1.8.4 Remarcado de una línea cualquiera

Es posible indicar a SynEdit que remarque una o varias líneas del contenido. Para ello primero debemos crear un método en nuestro formulario que identifique la línea a marcar y asigne los atributos:

```
procedure TForm1.SynEdit1SpecialLineMarkup(Sender: TObject;
                    Line: integer;
                    var Special: boolean;
                    Markup: TSynSelectedColor);
begin
  if Line = 2 then begin
     Special := True ;  //marca como línea especial
     Markup.Background := clGreen; //color de fondo
  end;
end;
```

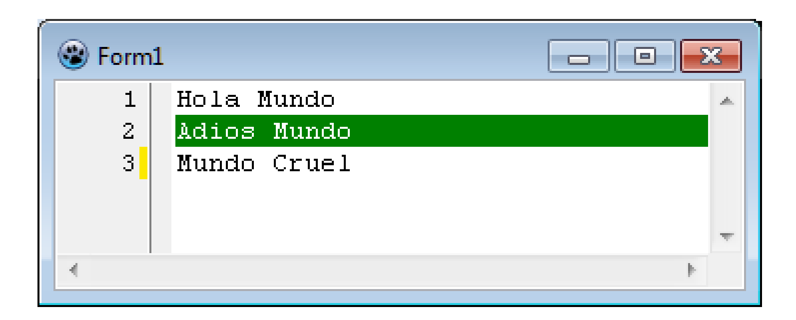
El tipo "TSynSelectedColor" está definido en la unidad "SynEditMiscClasses", así que será necesario incluirla antes, para trabajar.

Luego debemos asignar este método al evento "OnSpecialLineMarkup", del editor:

```
SynEdit1.OnSpecialLineMarkup := @SynEdit1SpecialLineMarkup;
```

Cada vez que el editor explora una línea, llama al evento "OnSpecialLineMarkup", para ver si tiene algún atributo especial o es una línea común. La implementación de este evento, debe ser de respuesta rápida para evitar interferir en el funcionamiento de SynEdit.

El resultado de este código sería algo como esto:



En este caso hemos activado el resaltado de la segunda línea, y permanecerá así marcada hasta que anulemos el evento. La línea marcada será siempre la segunda, aun cuando se inserten o eliminen líneas. Para marcar una línea que siga al texto contenido, se debe agregar un procesamiento adicional.

Si se cambia la línea remarcada por código, se debe refrescar al editor para que actualice la nueva línea marcada. Esto se puede hacer llamando al método "SynEdit1.Refresh" pero se recomienda usar método "SynEdit1.Invalidate", que es más eficiente.

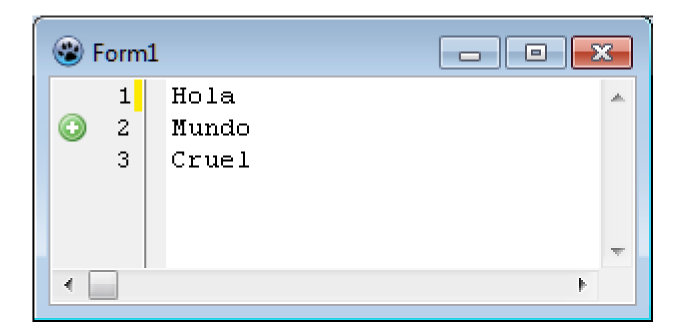
Existe además, el evento OnSpecialLineColors() que permite realizar una tarea similar, pero solo permite cambiar el color de fondo y el color del texto. Este evento está definido como:

```
TSpecialLineColorsEvent = procedure(Sender: TObject; Line: integer;
    var Special: boolean; var FG, BG: TColor) of object;
```

1.8.5 Marcadores de texto

Los marcadores de texto son ubicaciones especiales fijadas en el contenido de SynEdit. Podría decirse que son el equivalente digital de la cinta (separador) que sirve para marcar una página en un libro.

Sin embargo los marcadores de SynEdit, guardan posiciones que incluyen fila y columna, y puede haber tantos como se quiera. Los marcadores de texto se pueden mostrar gráficamente como un ícono en el Panel Vertical:



Los marcadores de texto, a diferencia de los vistos anteriormente, no resaltan el contenido del editor, sino que almacenan una ubicación y tienen la opción de mostrar un ícono en el panel vertical del editor.

Estos marcadores son útiles para guardar las posiciones de ciertas líneas de texto. Si se insertan o se eliminan líneas, el marcador tratará de mantener su ubicación, siguiendo a su línea.

Para crear un marcador, se debe incluir la unidad SynEditMarks y se debe tener una lista de íconos, almacenados en un TImageList. El siguiente código, usado para la figura anterior, crea un marcador y lo hace visible en la línea 2:

```
uses ... , SynEditMarks;

var
  marca: TSynEditMark;
...

marca := TSynEditMark.Create(SynEdit1);  //crea marcador
marca.ImageList:=ImageList1;    //asigna lista de íconos
marca.ImageIndex:=1;            //elige su ícono
marca.Line:=1;                  //define la línea donde se ubicará
marca.Visible:=true;            //lo hace visible
SynEdit1.Marks.Add(marca);      //agrega el marcador al editor
```

De esta forma se pueden ir agregando diversos marcadores en diversas posiciones de la pantalla, con el mismo o diferente ícono.

Es importante que la línea donde se ubique el marcador sea válida, o de otra forma no aparecerá de ningún modo.

Los marcadores mostrados, se manejan simplemente como posiciones especiales en el texto, sin ningún orden en especial.

Existen, sin embargo, otro tipo de marcadores llamados “BookMark”, que ofrecen además un número que los puede identificar.

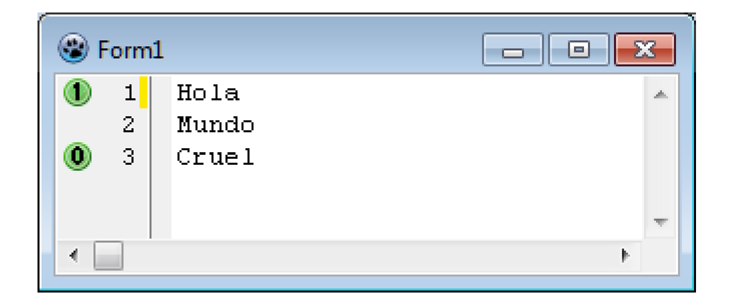
Los “BookMark” se suelen crear de forma distinta. Igualmente se requiere una lista de imágenes:

```
SynEdit1.BookMarkOptions.BookmarkImages := ImageList1;
SynEdit1.SetBookMark(0, 1, 1);  //BookMark 0 en la línea 1
```

```
SynEdit1.SetBookMark(1, 1, 3);  //BookMark 1 en la línea 3
```

El primer parámetro de SetBookMark() es el número de "BookMark" creado. Ese número lo identificará de ahí en adelante. Los otros dos parámetros son la fila y la columna que marcan. A pesar de que el ícono parece seleccionar a una fila, los "BookMark" guardan en realidad las coordenadas X-Y.

Aunque los íconos usados para los "BookMark", pueden ser de cualquier tipo, lo usual es usar íconos que se asocien al número del marcador:



Para eliminar un "BookMark", se debe conocer su número:

```
SynEdit1.ClearBookMark(5);
```

Este código elimina el "BookMark", número 5. Si el número de "BookMark" no existe, se ignora el comando.

Para acceder a un marcador cualquiera (puede ser "BookMark" o no), se hace como si fuera un elemento de una matriz:

```
marca := SynEdit1.Marks[0];
if marca.IsBookmark then ...
```

La propiedad "IsBookMark" permite identificar si un marcador es un "BookMark".

También se puede crear un "BookMark" como un marcador cualquiera:

```
uses ... , SynEditMarks;

var
  marca: TSynEditMark;
...

SynEdit1.BookMarkOptions.BookmarkImages := ImageList1;
marca := TSynEditMark.Create(SynEdit1) ;
marca.Line := 1;
marca.Column := 5;
marca.ImageIndex := 0;
marca.BookmarkNumber := 1;
marca.Visible := true;
marca.InternalImage := false;

SynEdit1.Marks.Add(marca);
```

La propiedad “InternalImage”, no es funcional en la versión actual de Lazarus. Es una reminiscencia de una facilidad que permitía usar íconos internos para los marcadores de texto.

Otra forma de crear un “BookMark” es agregándolo como un comando:

```
uses ... , SynEditKeyCmds;

SynEdit1.BookMarkOptions.BookmarkImages := ImageList1;
SynEdit1.ExecuteCommand(ecSetMarker4,#0,nil);
```

Existen 10 constantes definidas para los “BookMark”: ecSetMarker0 .. ecSetMarker9.

También existen 10 constantes predefinidas para cambiar el estado del “BookMark”: ecToggleMarker0 .. ecToggleMarker9 .

Por defecto, se crean atajos para agregar marcadores (BookMark), en SynEdit, al momento de agregar el control. Estos atajos son del tipo <Shift>+<Ctrl>+<número de marcador>.

Si se intentara agregar un marcador, sin tener una lista de íconos asignada, se generará un error en tiempo de ejecución.

Para deshabilitar los marcadores, se puede ocultar la parte correspondiente a los marcadores, en el panel vertical (Gutter), o se puede eliminar también los atajos de teclado (Propiedad “Keystrokes”).

1.8.6 Más sobre marcadores

Los marcadores más básicos en SynEdit, son objetos de la clase “TsynEditMarkup”. Todos los marcadores derivan directa o indirectamente de “TsynEditMarkup”.

La clase “TsynEditMarkup”, es una clase más o menos extensa, y se encuentra definida en la unidad “SynEditMarkup”. Sus campos públicos son:

```
  TSynEditMarkup = class(TObject)
  private
...
  protected
...
  public
    constructor Create(ASynEdit : TSynEditBase);
    destructor Destroy; override;
    Procedure PrepareMarkupForRow(aRow : Integer); virtual;
    Procedure FinishMarkupForRow(aRow : Integer); virtual;
    Procedure EndMarkup; virtual;
    Function  GetMarkupAttributeAtRowCol(const aRow: Integer;
const aStartCol: TLazSynDisplayTokenBound;
const   AnRtlInfo:   TLazSynDisplayRtlInfo)   :   TSynSelectedColor;   virtual;
abstract;
```

```
    Procedure GetNextMarkupColAfterRowCol(const aRow: Integer;
const aStartCol: TLazSynDisplayTokenBound;
const AnRtlInfo: TLazSynDisplayRtlInfo;
out   ANextPhys, ANextLog: Integer); virtual; abstract;
    procedure MergeMarkupAttributeAtRowCol(const aRow: Integer;
const aStartCol, AEndCol :TLazSynDisplayTokenBound;
const AnRtlInfo: TLazSynDisplayRtlInfo;
AMarkup: TSynSelectedColorMergeResult); virtual;

    // Notifications about Changes to the text
    Procedure TextChanged(aFirstCodeLine, aLastCodeLine, ACountDiff:
Integer); virtual; // 1 based
    Procedure TempDisable;
    Procedure TempEnable;
    procedure IncPaintLock; virtual;
    procedure DecPaintLock; virtual;
    function  RealEnabled: Boolean; virtual;

    property MarkupInfo : TSynSelectedColor read fMarkupInfo;
    property FGColor : TColor read GetFGColor;
    property BGColor : TColor read GetBGColor;
    property FrameColor: TColor read GetFrameColor;
    property FrameStyle: TSynLineStyle read GetFrameStyle;
    property Style : TFontStyles read GetStyle;
    property Enabled: Boolean read GetEnabled write SetEnabled;
    property Lines : TSynEditStrings read fLines write SetLines;
    property Caret : TSynEditCaret read fCaret write SetCaret;
    property TopLine : Integer read fTopLine write SetTopLine;
    property LinesInWindow : Integer read fLinesInWindow write
SetLinesInWindow;
    property InvalidateLinesMethod : TInvalidateLines write
SetInvalidateLinesMethod;
  end;
```

Para crear un marcador nuevo debemos crear un objeto de tipo “TsynEditMarkup” o de alguno de sus descendientes.

Ya hemos visto cómo crear un marcador en la sección 1.8.2 - Remarcado de la palabra actual, cuando creamos un objeto de la clase “TSynEditMarkupHighlightAllCaret”, porque esta clase deriva de “TsynEditMarkup”. También la clase “TSynEditMarkupHighlightAll”, usada para marcar una palabra cualquiera, es una descendiente de “TsynEditMarkup”.

El componente SynEdit, viene con varios marcadores, definidos por defecto. Una exploración de ellos se logra iterando sobre la tabla Markup[]:

```
  //itera por los marcadores
  for i:= 0 to SynEdit1.MarkupCount-1 do begin
    tmp := SynEdit1.Markup[i].MarkupInfo.StoredName;  //lee nombre
    ShowMessage(tmp);
  end;
```

La mayoría de marcadores predefinidos en SynEdit, tienen su campo “StoredName” vacío, así que el lazo anterior mostrará cadenas nulas, pero si hubiéramos creado un marcador con nombre, podríamos ubicarlo de esta forma.

El siguiente ejemplo muestra como deshabilitar un marcador creado con el nombre “ResPalabraActual”:

```
var
  marc: TSynEditMarkup;
  ...

  marc := nil;
  //busca marcador por nombre
  for i:= 0 to ed.MarkupCount-1 do begin
    tmp := ed.Markup[i].MarkupInfo.StoredName;
    if ed.Markup[i].MarkupInfo.StoredName='ResPalabraActual' then
      marc := ed.Markup[i];
  end;
  if marc<>nil then begin  //hay marcador
     marc.Enabled:=false;  //desactiva
  end;
```

Otra forma de ubicar un marcador en SynEdit es usar el método MarkupByClass[], que permite especificar el nombre de la clase del marcador que queremos encontrar:

```
   marc := SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret];
```

Si hubiera dos marcadores de la misma clase, se retornará el primero de ellos.

El siguiente código, muestra cómo definir el color del borde del texto remarcado, por el marcador TSynEditMarkupHighlightAllCaret:

```
var
  marc: TSynEditMarkup;
  ...
  marc := SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret];
  TSynEditMarkupHighlightAllCaret(marc).MarkupInfo.FrameColor := clGreen;
```

Para acceder a las propiedades visuales del texto remarcado, se usa la propiedad “MarkupInfo” del resaltador.

La propiedad “MarkupInfo”, es un descendiente de la clase “TSynHighlighterAttributes” (Ver Sección 2.1.1 - Conceptos Claves), por lo tanto se comporta como un contenedor de las propiedades de apariencia de un texto.

Al crear un marcador es conveniente darle un nombre y guardarlo en la propiedad “StoredName”, de “MarkupInfo”, para poder ubicarlo luego desde el editor.

Los marcadores vistos en las secciones anteriores, que resaltan la apariencia del texto mostrado, son relativamente simples (remarcado de una palabra o de una línea). Sin embargo los marcadores de SynEdit son bastante elaborados, y permiten implementar casos de marcas complejas del texto.

Un ejemplo de marcadores más completos, son los que usa el editor de Lazarus, que permiten resaltar el inicio y fin de bloques de código como BEGIN-END o REPEAT-UNTIL.

Para crear estos marcadores elaborados, se debe acceder a campos protegidos de SynEdit, así que se debe crear una clase que derive de SynEdit y crear nuestro editor a partir de esa clase. Una vez definido nuestro editor, tendremos la posibilidad de crear cualquier marcador, cuya lógica que debe ser definida por código.

El siguiente código crear un editor a partir de SynEdit, creando una clase descendiente de SynEdit (subclass), y define dos bloques de texto para remarcarlos:

```
unit Unit1; {$mode objfpc}{$H+}

interface
uses
  Classes, SysUtils, Forms, Controls, Graphics,
  SynEdit, SynEditMarkupSelection,
  SynEditTypes, SynEditPointClasses, SynEditMarkup;

type
  { TMiEditor }
  TMiEditor = class(TSynEdit)  //define la clase de mi Editor
  private
    Bloque1: TSynEditSelection;        //bloque de selección 1
    Markup1: TSynEditMarkupSelection;  //marcador 1

    Bloque2: TSynEditSelection;        //bloque de selección 2
    Markup2: TSynEditMarkupSelection;  //marcador 2
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

  { TForm1 }
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    MiEditor: TMiEditor;
  end;

var
  Form1: TForm1;
```

```
implementation
{ TMiEditor }
constructor TMiEditor.Create(AOwner: TComponent);
var
  MarkupManager: TSynEditMarkupManager;
begin
  inherited Create(AOwner);
  MarkupManager   :=   TSynEditMarkupManager(MarkupMgr);      //MarkupMgr   es
"TObject"
  //crea un bloque para remarcado
  Bloque1 := TSynEditSelection.Create(ViewedTextBuffer, false);
  Bloque1.InvalidateLinesMethod := @InvalidateLines;
  Markup1 := TSynEditMarkupSelection.Create(self, Bloque1);
  MarkupManager.AddMarkUp(Markup1);   //agrega marcador
  //crea un bloque para remarcado
  Bloque2 := TSynEditSelection.Create(ViewedTextBuffer, false);
  Bloque2.InvalidateLinesMethod := @InvalidateLines;
  Markup2 := TSynEditMarkupSelection.Create(self, Bloque2);
  MarkupManager.AddMarkUp(Markup2);   //agrega marcador
end;

destructor TMiEditor.Destroy;
begin
  Bloque1.Free;    //Markup1, se destruye con SynEdit
  Bloque2.Free;    //Markup2, se destruye con SynEdit
  inherited Destroy;
end;

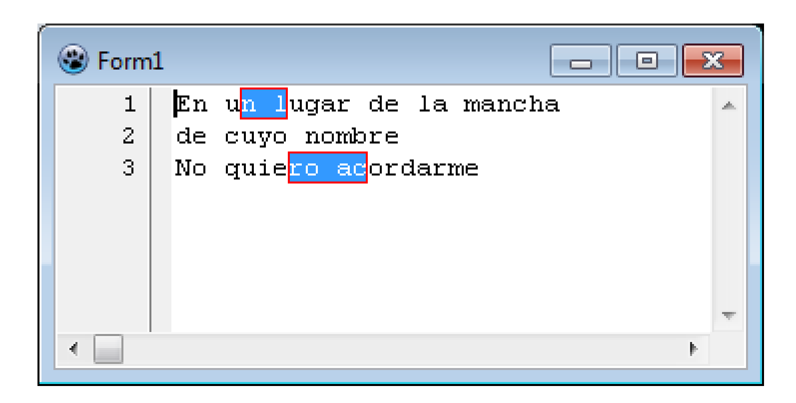
{$R *.lfm}
{ TForm1 }
procedure TForm1.FormCreate(Sender: TObject);
begin
  MiEditor := TMiEditor.Create(self);  //crea mi componente Editor
  MiEditor.Parent:= self;          //lo ubica en el formualrio
  MiEditor.Align := alClient;      //lo alinea
  //escribe un texto
  MiEditor.Lines.Add('En un lugar de la mancha');
  MiEditor.Lines.Add('de cuyo nombre');
  MiEditor.Lines.Add('No quiero acordarme');
  //define sección 1 a remarcar
  MiEditor.Bloque1.StartLineBytePos := Point(5,1);
  MiEditor.Bloque1.EndLineBytePos := Point(8,1);
  MiEditor.Markup1.Enabled := True;
  MiEditor.Markup1.MarkupInfo.FrameColor := clRed;

  //define sección 2 a remarcar
  MiEditor.Bloque2.StartLineBytePos := Point(8,3);
  MiEditor.Bloque2.EndLineBytePos := Point(13,3);
  MiEditor.Markup2.Enabled := True;
  MiEditor.Markup2.MarkupInfo.FrameColor := clRed;
```

```
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  MiEditor.Destroy;        //libera
end;
end.
```

El efecto en el texto será similar al mostrado en la siguiente figura:



Para este remarcado, se ha usado la clase "TSynEditSelection", definida en la unidad "SynEditPointClasses", que permite especificar áreas de selección dentro del editor. Por defecto tiene el mismo color de fondo que la selección, pero puede cambiarse con la propiedad "MarkupInfo.Background".

Una observación en este remarcado, es que las áreas resaltadas tienen coordenadas fijas, aunque se modifique el texto. Si se desea que el remarcado, siga al texto, se debe implementar por código, la lógica necesaria.

1.9 Resumen de Propiedades y Métodos

PROPIEDAD	DESCRIPCIÓN
BeginUpdate, EndUpdate	Deshabilita y Habilita (respectivamente) el refresco de pantalla de SynEdit. Ver 1.5.4 - Hacer y Deshacer.
BeginUndoBlock EndUndoBlock	Permiten agrupar varias acciones de cambio, para deshacerse con una sola llamada a Undo. Ver 1.5.4 - Hacer y Deshacer.
BlockBegin BlockEnd	Indican las coordenadas del texto seleccionado. Ver Sección 1.6 - Manejo de la selección
BlockIndent	Indica la cantidad de espacios que se usan para indentar un bloque de texto, cuando se ejecutan las acciones de indentación de bloques (En la IDE de Lazarus están mapeadas como Ctrl+U y Ctrl+I).
BookMarkOptions	Es un conjunto de opciones que permiten configurar los marcadores de texto.
BracketHighlightStyle	Configura el comportamiento del resaltado de los delimitadores paréntesis, llaves y corchetes. Indica cómo se determina el resaltado. Pueden ser: sbhsBoth sbhsLeftOfCursor sbhsRightOfCursor
BracketMatchColor	Atributo usado para resaltar los delimitadores paréntesis, llaves y corchetes.
CanUndo	Indica si hay acciones por deshacer en el editor
CanRedo	Indica si hay acciones por rehacer en el editor
CanPaste	Indica si hay contenido para pegar en el editor
CaretX	Permiten leer o fijar la coordenada horizontal del cursor del editor. El primer carácter tiene coordenada 1.
CaretY	Permiten leer o fijar la coordenada vertical del cursor del editor. La primera fila tiene coordenada 1.
CaretXY	Permiten leer o fijar las coordenadas X, Y del cursor del editor. Ver Sección 1.4.1 - Coordenadas del editor
ClearAll	Borra todo el contenido del editor.
ClearSelection	Borra el texto seleccionado.
ClearUndo	Limpia y reinicia la memoria de "deshacer". Una vez ejecutado, ya no se podrán deshacer los cambios.
Color	Color de fondo del editor.
CopyToClipboard	Copia el texto seleccionado al portapapeles. Ver Sección 1.5.3 - El Portapapeles.
CutToClipboard	Corta el texto seleccionado al portapapeles. Ver Sección 1.5.3 - El Portapapeles.
ExecuteCommand	Envía un comando al editor. Ver sección 1.5.1 - Ejecutar comandos
ExtraCharSpacing	Indica el espaciado que hay entre las letras. Por defecto es cero. Ver Sección 1.3.3 - Tipografía.
ExtraLineSpacing	Indica el espaciado que hay entre líneas. Por defecto es cero. Ver Sección

	1.3.3 - Tipografía.
FoldAll	Permite cerrar todos los bloques de plegado que existan en un editor. También puede plegar por niveles.
Font	Objeto que define las propiedades de la tipografía a usar en el editor. Tiene diversas propiedades, como el nombre de la fuente, el tamaño, el juego de caracteres, etc. Ver Sección 1.3.3 - Tipografía
Gutter	Referencia al objeto que define el panel lateral del editor, donde suele parecer el número de línea.
GetHighlighterAttriAtRowCol	Lee el token y el atributo para una posición específica del texto. Solo es válido cuando se tiene un resaltador asociado al editor.
Highlighter	Referencia al resaltador que se va a usar para implementar el resaltado de sintaxis (Ver Sección 2.3 - Coloreado de Sintaxis Usando Código).
InsertMode	Permite pasar del modo normal al modo INSERT, en donde se sobre-escriben los caracteres ingresados. Cuando se pone a FALSE, se entra a modo de inserción.
InsertTextAtCaret	Inserta un texto en la posición actual del cursor. Ver Sección 1.5 - Modificar el contenido
Keystrokes	Almacena los atajos de teclado, y los comandos a los cuales están asociados esos atajos de teclado.
LineHighlightColor	Es el color de fondo de la línea en la que se encuentra el cursor en ese momento. Ver sección 1.8.3 - Remarcado de la línea actual
LinesInWindow CharsInWindow	Indica la cantidad de Líneas y columnas visibles en pantalla. Depende únicamente del tamaño de la pantalla y el espaciado entre caracteres y líneas.
Lines	Lista de todo el contenido del editor. Es una lista de cadenas (similar a TStringList), cada ítem representa una línea. Empieza en el elemento 0.
LineText	Almacena siempre el contenido de la línea actual.
MaxLeftChar	Limita la posición horizontal del cursor cuando se encuentra en modo “Cursor flotante” - Ver sección 1.4.2 - Manejo del cursor
MaxUndo	Número máximo de operaciones a deshacer.
Modified	Indica cuando el contenido del editor ha sido modificado. También puede escribirse.
MoveCaretIgnoreEOL	Posiciona el cursor sin considerar los límites de la línea destino.
Options	Diversas opciones adicionales para configurar al editor, como el indentado automático, el comportamiento del cursor fuera de los límites de la línea, o la conversión de las tabulaciones en espacios. Ver sección 1.9.1 - Propiedad Options
PasteFromClipboard	Pega el texto del portapapeles en la posición de cursor. Ver Sección 1.5.3 - El Portapapeles.
Redo	Vuelve a realizar una acción desecha con “Undo”. Ver Sección 1.5.4 - Hacer y Deshacer.
RightEdge	Indica la posición de la línea vertical (margen derecho) que se usa para marcar el límite del área de impresión. Ver sección 1.3 - Apariencia
RightEdgeColor	Color del margen derecho.
SelectAll	Selecciona todo el contenido del editor

SelText	Propiedad que permite leer o modificar el texto seleccionado. Ver Sección 1.6 - Manejo de la selección
SelectionMode	Modo de selección del cursor. Permite pasar cambiar el modo de selección normal del texto.
SearchReplace	Permite realizar búsqueda y reemplazo de texto dentro del editor. Ver Sección 1.7 - Búsqueda y Reemplazo
SelectWord	Selecciona la palabra actual en donde se encuentra el cursor. Ver Sección 1.6 - Manejo de la selección.
SelectLine	Selecciona la línea actual en donde se encuentra el cursor. Ver Sección 1.6 - Manejo de la selección.
SelectParagraph	Selecciona el párrafo actual en donde se encuentra el cursor. Ver Sección 1.6 - Manejo de la selección.
SelectToBrace	Selecciona el bloque delimitado por paréntesis, llaves o corchetes. Ver Sección 1.6 - Manejo de la selección.
ScrollBars	Controla la visibilidad de las barras de desplazamiento horizontal y vertical.
TabWidth	Es la cantidad de espacios que se usan para representar a una tabulación. Esta opción puede ignorarse en el modo de Tabulación Inteligente (Options.eoSmartTabs = TRUE)
Undo	Deshace la última acción realizada sobe el editor. Ver Sección 1.5.4 - Hacer y Deshacer.
UnfoldAll	Permite expandir todos los bloques de plegado que se hayan cerrado.

1.9.1 Propiedad Options y Options2

Existen diversos parámetros que se pueden configurar mediante la propiedad “Options” de SynEdit. La sintaxis común es:

```
SynEdit1.Options := [eoKeepCaretX, eoTabIndent, ... ];
```

Esta propiedad es un conjunto de tipo “TSynEditorOptions” que puede incluir los siguientes elementos:

VALOR	DESCRIPCIÓN
eoAutoIndent	Posiciona el cursor en la nueva línea con la misma cantidad de espacios que la línea anterior.
eoBracketHighlight	Resaltará los delimitadores paréntesis, llaves y corchetes (brackets).
eoEnhanceHomeKey	Tecla “Home” saltará al inicio de la línea si está más cercana (similar a Visual Studio)
eoGroupUndo	Agrupa todos los cambios del mismo tipo, en una sola acción de Deshacer/Rehacer, en lugar de manejar cada comando separadamente.
eoHalfPageScroll	Los desplazamientos página-arriba o página-abajo, solo saltarán media página a la vez.
eoHideRightMargin	Oculta la línea del margen derecho.
eoKeepCaretX	Limita la posición del cursor para moverse, solo hasta el final de la línea. De otra forma puede pasar hasta más allá de la línea. Ver Sección 1.4.2 - Manejo del cursor
eoNoCaret	Hace el cursor invisible.
eoNoSelection	Deshabilita la selección del texto.
eoPersistentCaret	No oculta el cursor cuando se pierde el foco.
eoScrollByOneLess	Los desplazamientos página-arriba o página-abajo, serán con una línea menos.
eoScrollPastEof	Permiten al cursor pasar más allá de la marcar de fin de archivo.
eoScrollPastEol	Permite al cursor pasar más allá del último carácter de una línea. Funciona aun cuando eoKeepCaretX está presente.
eoScrollHintFollows	El desplazamiento de la etiqueta (Hint) sigue al desplazamiento del mouse al moverse verticalmente.
eoShowScrollHint	Muestra una etiqueta (Hint) con el número de la primera línea visible, al hacer desplazamiento vertical.
eoShowSpecialChars	Muestra caracteres especiales.
eoSmartTabs	Al tabular el cursor se ubicará en la posición del siguiente espacio blanco de la línea previa.
eoTabIndent	Las teclas <Tab> y <Shift><Tab> funcionarán para indentar o quitar la indentación de un bloque seleccionado.
eoTabsToSpaces	Hace que la tecla <Tab> inserte espacios (especificado en la propiedad “TabWidth”) en lugar del carácter TAB. No convierte

	los caracteres de tabulación que ya existen, simplemente evita que se agreguen más.
eoTrimTrailingSpaces	Los espacios al final de las líneas serán recortados y no se guardarán.

Adicionalmente a “Options”, se pueden incluir otros valores en la propiedad “Options2”:

VALOR	DESCRIPCIÓN
eoCaretSkipsSelection	El cursor saltará sobre la selección al usar las direccionales derecha e izquierda.
eoCaretSkipTab	El cursor saltará todos los espacios que componen una tabulación. Sin esta opción, el cursor pasar por la tabulación como si fueran espacios.
eoAlwaysVisibleCaret	Mueve el cursor de modo que siempre sea visible al hacer desplazamientos de pantalla.
eoEnhanceEndKey	Al pulsar la tecla <End>, mueve al final de la línea, pero sin considerar los espacios finales.
eoFoldedCopyPaste	Mantiene las propiedades de plegado en operaciones de copiado/pegado.
eoPersistentBlock	Mantiene los bloques de selección, aun cuando el cursor esté fuera del bloque.
eoOverwriteBlock	No mantiene los bloques persistentes. Los sobre-escribe en operaciones Insert/Delete.
eoAutoHideCursor	Permite ocultar el cursor con operaciones del teclado.

2 Resaltado de Sintaxis y Autocompletado con SynEdit.

2.1 Introducción

El resaltado de sintaxis es la capacidad de poder dar atributos visuales distintos a cada elemento de un texto, de forma que lo haga más legible. Estos atributos solo se muestran en el editor más no forman parte del texto, cuando se guardan en un archivo.

La siguiente figura muestra un trozo de código en Pascal, mostrado en un editor simple y en otro con resaltado de sintaxis:

Form1

```
bolsa[Leidos] := #0; //agrega delimita
Pbolsa := @bolsa;    //cadena PChar
//Explora en busca de delimitadores de
Pos13 := Pos(#13, Pbolsa);
Pos10 := Pos(#10, Pbolsa);
if Pos13 = 0 then
   //solo hay separador #10 o ninguno
   if Pos10<>0 then
      Formato := TAR_UNIX     //solo h
   else
      Formato := TAR_DESC  //no se enc
```

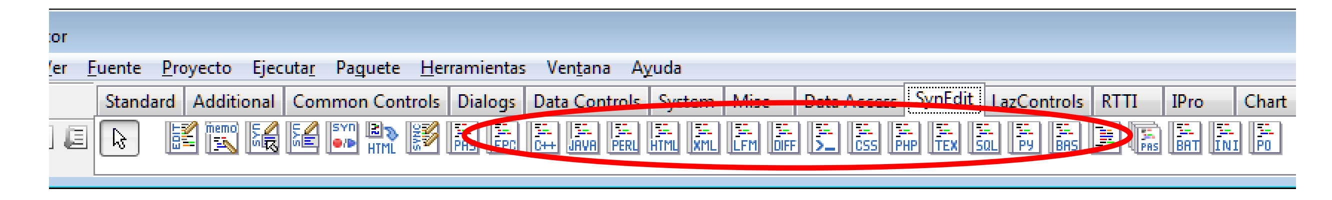
Form1

```
bolsa[Leidos] := #0; //agrega delimita
Pbolsa := @bolsa;    //cadena PChar
//Explora en busca de delimitadores de
Pos13 := Pos(#13, Pbolsa);
Pos10 := Pos(#10, Pbolsa);
if Pos13 = 0 then
   //solo hay separador #10 o ninguno
   if Pos10<>0 then
      Formato := TAR_UNIX     //solo h
   else
      Formato := TAR_DESC   //no se enc
```

La diferencia es evidente. Un código con resaltado de sintaxis, es más fácil de entender y leer. Sin embargo, este resaltado, debe hacerse de manera consistente y sencilla, de otra forma podría confundir, en lugar de ayudar.

El resaltado de sintaxis, es también llamado "Coloreado de sintaxis", por el uso común del color como forma de diferenciar los elementos del texto, pero el color es solo uno de los atributos que pueden ser usados, en un contexto más general, que puede incluir por ejemplo el tipo de letra. Para generalizar el uso de atributos diversos, es que usamos el término "Resaltado de Sintaxis", pero también usaremos "Coloreado de Sintaxis", como un sinónimo.

El componente TSynEdit no incluye resaltadores incorporados en el mismo control, sino que debe estar asociado a un resaltador externo. Para ello el paquete SynEdit, incluye diversos componentes de tipo resaltador (de diversos lenguajes), que pueden ser usados con SynEdit.



También podemos crear nuestro propio resaltador, si es que no encontramos uno que se adapte a nuestras necesidades.

2.1.1 Conceptos Claves

En la Sección 1.4.1 - Coordenadas del editor, vimos como SynEdit, interpreta los bytes de datos para llegar al nivel de celdas de pantalla. Y que por lo general un carácter ocupará una celda.

Para implementar el resaltado de sintaxis, es necesario subir el nivel de análisis. Los caracteres no son apropiados para determinar el resaltado (a menos que se quiera un resaltado carácter por caracter), es mejor usar el concepto de “token”.

Primero revisemos algunos conceptos importantes, antes de avanzar con el tema:

- **Carácter**.- Son los componentes visuales mínimos que componen un texto. Obedecen a una codificación definida (UTF-8 en el caso de SynEdit). Permiten definir elementos de mayor tamaño como los tokens. Por lo general ocupan una celda en la pantalla, pero podrían ocupar dos.

- **Token**.- Es la unidad de información compuesta de uno o más caracteres, que se agrupan por reglas específicas, y que tienen sentido en una sintaxis específica. Los tokens son las partes en las que se divide un lenguaje de programación, a nivel léxico.

- **Atributo**.- Son las propiedades de resaltado que se aplican a un token particular. En SynEdit, un atributo está representado por un objeto de la clase “TSynHighlighterAttributes” y permiten definir propiedades como el color del texto, el color del fondo, el tipo de letra, la activación de negrita, y otras propiedades más.

El resaltado de sintaxis empieza en el nivel de los tokens y cada token puede tener atributos diferentes del resto de tokens. Se puede decir que la unidad mínima de información para los resaltadores de sintaxis en SynEdit es el token.

El token es la unidad mínima de información para el resaltado de sintaxis en el componente SynEdit

La siguiente figura ayudará a aclarar la idea:

bytes	7B	78	3D	31	32	3B	63	61	64	3D	22	74	C3	BA	22	7D	3B	32	32	2F	2F	63	6F	6D	65	6E	74
caracteres	{	x	=	1	2	;	c	a	d	=	"	t	ú		"	}	;			/	/	c	o	m	e	n	t
celdas	{	x	=	1	2	;	c	a	d	=	"	t	ú		"	}	;			/	/	c	o	m	e	n	t
tokens	simbolo	identificador	simbolo	número		simbolo	identificador			simbolo	cadena					simbolo	simbolo	espacio		comentario							

Como se puede ver, los tokens pueden ser de diversos tipos, y los atributos se deben aplicar a cada tipo de token (no a cada token). Así, todas las cadenas deben tener los mismos atributos en el mismo texto.

Los tipos comunes de tokens son:

- Identificador.- Algunos de ellos pueden ser palabras claves
- Número.- Pueden tener diversa representación.
- Espacio.- Todos los caracteres que no son visibles
- Cadena.- Por lo general encerradas entre comillas simples o dobles.
- Comentario.- Pueden ser de una o varias líneas.
- Símbolo.- Símbolos diversos como operadores.
- De Control.- Puede incluir saltos de línea o marcas de fin.

La forma como se agrupan los caracteres en tokens (y sus tipos), no está pre-definida por SynEdit. No hay reglas establecidas. Cuando SynEdit realiza el resaltado de sintaxis, hace uso de un resaltador, y es este resaltador, el que define la forma de identificar a los tokens en el texto.

Los resaltadores predefinidos, incluyen ya estas definiciones, pero si creamos un resaltador personalizado, debemos definir nosotros estas reglas.

Simplificando el procesos de resaltado de sintaxis, podríamos decir que consiste en, separar el texto en tokens y darle color (atributo) a cada token. Esta descripción simplista es, sin embargo, muy cierta, pero el proceso es más complicado de lo que suena. La mayor dificultad estriba en identificar de forma precisa y bastante rápida a cada uno de los tokens.

Los tokens son los elementos básicos con los que trataría un analizador léxico (lexer). Un resaltador de sintaxis es, en cierta forma, un analizador léxico, pero su objetivo no es servir de base para un posterior análisis sintáctico o semántico, sino que bastará con identificar los tokens y sus tipos para facilitar el resaltado[6].

[6] La posibilidad de convertir a un resaltador en un analizador sintáctico, queda a libertad del programador. Los resaltadores pre-definidos en Lazarus solo identifican tokens y en algunos casos, delimitadores de bloques como BEGIN ..END (para el plegado), pero no se diseñan con el fin de servir de "lexers". Hacerlos de esta forma, involucraría un procesamiento mayor y resultaría en un análisis más lento, que va en contra del objetivo de un resaltador.

2.2 Coloreado de Sintaxis Usando Componentes Predefinidos

Este tipo de coloreado de sintaxis implica el uso de componentes predefinidos de sintaxis. Estos componentes vienen preparados para trabajar en un lenguaje predefinido.

2.2.1 Usando un lenguaje predefinido

Estos componentes de sintaxis vienen ya instalados en el entorno Lazarus. Existen componentes para la mayoría de lenguajes comunes como: Pascal, C++, Java, HTML, Phyton, etc.

El método es simple: Arrastramos el control al formulario, donde está el control SynEdit que vamos a usar. Luego lo enlazamos, usando la propiedad “highlighter”, del SynEdit.

Posteriormente podemos elegir los colores y atributos de las palabras claves, comentarios, números, y otras categorías, accediendo a las propiedades del objeto “TSynXXXSyn” (donde XXX representa el lenguaje elegido). Cada control “TSynXXXSyn”, está representado por una unidad.

El objeto “TSynXXXSyn” viene optimizado para trabajar en su lenguaje predefinido y responde bien en velocidad. Sin embargo, las palabras claves y su detección están empotradas (“hardcoded”) en el código de su unidad. Intentar agregar una palabra reservada más, no es tan sencillo, por la forma como se optimiza la búsqueda de palabras claves.

El uso de componentes predefinidos nos ahorra todo el trabajo de tener que procesar una sintaxis completa de un lenguaje conocido.

2.2.2 Usando un lenguaje personalizado

Adicionalmente a los componentes de los lenguajes predefinidos, existe un componente que se puede personalizar para un lenguaje diferente. Este es el componente “TSynAnySyn”.

Para usar este componente, solo basta con incluirlo en el formulario, como cualquier otro componente de sintaxis. Pero antes de usarlo se le debe indicar cuales son las palabras claves que componen su sintaxis.

SI bien este componente soporta la definición de varios lenguajes, simples, no permite mucha flexibilidad a la hora de manejar los comentarios.

Además este componente no tiene un desempeño bueno en cuanto a velocidad, porque su diseño multi-lenguajes, no permite optimizarlo adecuadamente. No es recomendable el uso de este componente con un lenguaje de muchas palabras claves o en textos grandes.

Si se desea implementar un lenguaje nuevo con un buen desempeño y muy personalizado, es recomendable hacerlo por código.

2.3 Coloreado de Sintaxis Usando Código

Este tipo de coloreado permite personalizar de forma eficiente una sintaxis específica. Debe usarse cuando el lenguaje a usar no existe ya, en uno de los componentes predefinidos, o no cumple con el comportamiento requerido.

Antes de diseñar nuestra clase para el manejo del coloreado, debemos entender un poco el funcionamiento del coloreado de sintaxis:

Para el coloreado de sintaxis, se debe crear una clase descendiente de "TSynCustomHighlighter", a esta clase la llamaremos clase resaltador, o clase sintaxis, de la cual instanciaremos un resaltador. Esta resaltador debe asociarse luego al editor SynEdit, que implementará el coloreado de sintaxis.

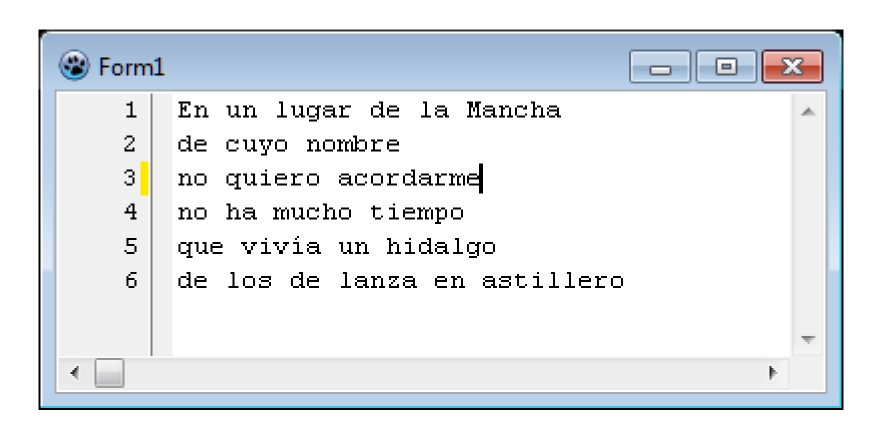
Cuando se asocia un resaltador, a un editor "TSynEdit", este empezará a llamar a las rutinas del resaltador, cada vez que requiera información sobre el coloreado de la sintaxis.

TSynEdit no guarda información del coloreado del texto en alguna parte. Siempre que requiera información de coloreado, llamara al resaltador, para obtener, "sobre la marcha", la información del coloreado.

Este resaltador, debe implementar el análisis del texto y la identificación de los elementos del texto para dar la información de los atributos del texto a SynEdit.

La exploración del texto a colorear lo hace SynEdit, procesando línea por línea. Cuando se modifica, una línea, se repinta, la línea modificada y las siguientes líneas, a partir de la línea modificada, que se muestren en la ventana visible. Este comportamiento en normal si se considera que puede incluirse en una línea un elemento de sintaxis que afecte a las demás líneas del texto (como el inicio de un comentario de varias líneas).

Consideremos el caso de un editor con el siguiente texto:



Al mostrar la ventana, después de haber estado oculta, se generarán los siguientes eventos:

```
1. SetLine: En un lugar de la Mancha
2. SetLine: de cuyo nombre
3. SetLine: no quiero acordarme
4. SetLine: no ha mucho tiempo
5. SetLine: que vivía un hidalgo
6. SetLine: de los de lanza en astillero
```

La etiqueta “SetLine”, indica que se está explorando la línea mostrada.

Al modificar la línea número 3 del mismo texto (un cualquier parte), la secuencia de exploracoión cambia un poco:

```
1. SetLine: no quiero acordarme,
2. SetLine: no ha mucho tiempo
3. SetLine: que vivía un hidalgo
4. SetLine: de cuyo nombre
5. SetLine: no quiero acordarme,
6. SetLine: no ha mucho tiempo
7. SetLine: que vivía un hidalgo
```

Podemos ver que el editor hace una exploración de las siguientes dos líneas, y luego hace una exploración nuevamente, pero empezando una línea anterior.

2.3.1 Casos de Resaltado de Sintaxis.

Al realizar el resaltado de sintaxis, debemos considerar diversas situaciones. Se pueden diferenciar 3 casos:

- Resaltado sencillo de tokens. Es el coloreado común en el que ciertas palabras o identificadores claves del texto se ponen de un color determinado. Normalmente se definen diversas categorías como palabras reservadas, palabras claves, identificadores, variables, macros, etc. Se puede definir diversos atributos de texto, para cada categoría. Los tokens se identifican por los caracteres que pueden contener.
- Resaltado de comentarios de una línea. Este coloreado es típico de los comentarios de una línea de la mayoría de lenguajes. Implica poner de un color específico, el texto de un comentario, desde el inicio hasta el final de la línea.
- Resaltado de rangos de texto o de contexto. Este coloreado se aplica también a los comentarios, o las cadenas. El coloreado del texto, afecta a un grupo de palabras, que pueden estar en una misma línea u ocupar varias líneas consecutivas. Los tokens se identifican por sus delimitadores.

2.3.2 Exploración de líneas

Cada línea se asume que está dividida en elementos llamados “tokens”. No hay parte de una línea que no sea un token. Un token puede ser un identificador, un símbolo, un carácter de control, un carácter en blanco, etc.

Un token puede tener uno o más caracteres de longitud. Cada tokens o tipo de token, puede tener atributos particulares.

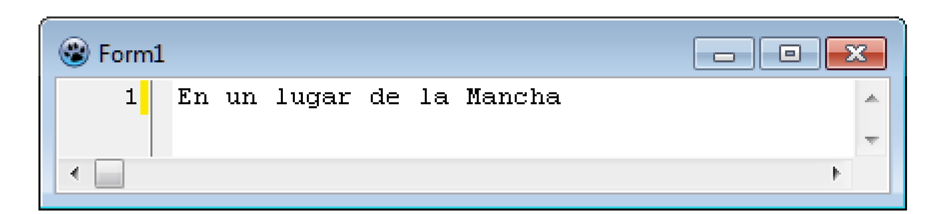
Cada vez que SynEdit necesita procesar una línea, se produce una secuencia de llamadas al resaltador (en realidad a "TSynCustomHighlighter", pero este los canaliza al resaltador que se esté usando), para obtener los atributos de cada elemento de la línea:

- Primeramente se llama el método "SetLine", para indicar que se está empezando la exploración de una nueva línea. Después de esta llamada espera que el método "GetTokenEx" o "GetTokenAttribute" devuelvan información sobre el token actual.
- Después de la llamada a "SetLine", se generarán múltiples llamadas al método "Next", para acceder a los siguientes tokens.
- SynEdit, espera que después de cada llamada a "Next", los métodos "GetTokenEx" y "GetTokenAttribute" devuelvan información sobre el token actualmente apuntado.
- Cuando SynEdit quiere verificar si se ha llegado al final de la línea de trabajo, hará una llamada a "GetEol". Esta debe estar funcional desde que se llama a "SetLine".

Las llamadas a estos método se producen repetidamente y en gran cantidad para cada línea. Por ello estos métodos, deben ser de respuesta rápida y de implementación eficiente. La demora en el procesamiento de alguno de estos métodos afectará al rendimiento del editor.

Para tener una idea del trabajo que hace SynEdit, en cuanto a coloreado de sintaxis, presentamos a continuación, la secuencia de métodos llamados cuando se muestra la ventana de editor, que estaba oculta.

La ventana ejemplo contiene este texto:



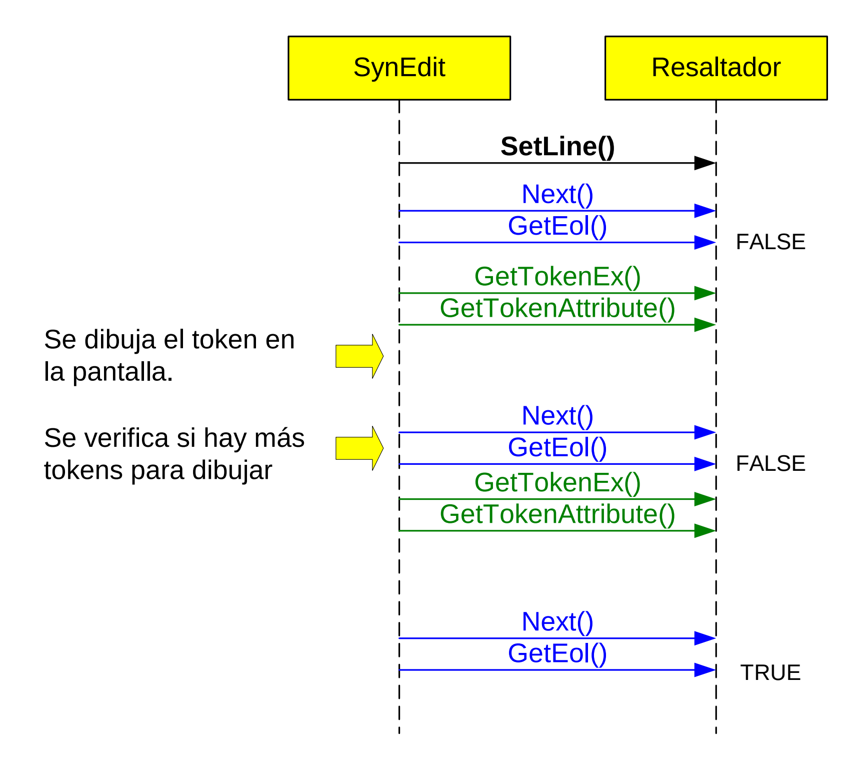
La secuencia de eventos llamados al mostrar esta ventana es:

```
1. SetLine: "En un lugar de la Mancha"
2.    Next:0
3.    GetEol
4.    GetTokenEx
5.    GetTokenAttribute
6.    Next:2
7.    GetEol
8.    GetTokenEx
9.    GetTokenAttribute
10.    Next:3
11.    GetEol
12.    GetTokenEx
13.    GetTokenAttribute
14.    Next:5
15.    GetEol
16.    GetTokenEx
17.    GetTokenAttribute
18.    Next:6
19.    GetEol
20.    GetTokenEx
21.    GetTokenAttribute
22.    Next:11
23.    GetEol
24.    GetTokenEx
25.    GetTokenAttribute
26.    Next:12
27.    GetEol
28.    GetTokenEx
29.    GetTokenAttribute
30.    Next:14
31.    GetEol
32.    GetTokenEx
33.    GetTokenAttribute
34.    Next:15
35.    GetEol
36.    GetTokenEx
37.    GetTokenAttribute
38.    Next:17
```

```
39.     GetEol
40.     GetTokenEx
41.     GetTokenAttribute
42.     Next:18
43.     GetEol
44.     GetTokenEx
45.     GetTokenAttribute
46.     Next:24
47.     GetEol
```

El valor indicado después del Evento Next, corresponde al carácter inicial que se explora. Se pude observar que el editor comprueba siempre si se ha llegado al final, después de cada llamada a "Next". Si después de una llamada a Next(), el editor obtiene TRUE en GetEol(), asumirá que está en el carácter final de la línea, y ya no pedirá información de atributo.

El siguiente diagrama de secuencia aclarará el proceso:



El editor va dibujando en pantalla, siempre token por token. El método GetTokenEx() devuelve la extensión del token, y el método GetTokenAttribute(), devuelve el atributo a aplicar al texto. Esa información es todo lo que necesita SynEdit para dibujar una porción del texto con atributos, en pantalla.

Esta secuencia Next() – GetEol() - GetTokenEx() - GetTokenAttribute() , se va repitiendo por toda la línea hasta que GetEol(), devuelva TRUE, lo que indicará que ya no quedan tokens por explorar en esa línea.

Si en nuestro texto de ejemplo realizamos una modificación, como insertar una coma al final de la línea, se genera la siguiente secuencia:

```
1. SetLine: "En un lugar de la Mancha"
2.      Next:0
3.      GetEol
4.      Next:2
5.      GetEol
6.      Next:3
7.      GetEol
8.      Next:5
9.      GetEol
10.     Next:6
11.     GetEol
12.     Next:11
13.     GetEol
```

```
14.    Next:12
15.    GetEol
16.    Next:14
17.    GetEol
18.    Next:15
19.    GetEol
20.    Next:17
21.    GetEol
22.    Next:18
23.    GetEol
24.    Next:24
25.    GetEol
26.    Next:25
27.    GetEol
28. SetLine: En un lugar de la Mancha,
29.    Next:0
30.    GetEol
31.    GetTokenEx
32.    GetTokenAttribute
33.    Next:2
34.    GetEol
35.    GetTokenEx
36.    GetTokenAttribute
37.    Next:3
38.    GetEol
39.    GetTokenEx
40.    GetTokenAttribute
41.    Next:5
42.    GetEol
43.    GetTokenEx
44.    GetTokenAttribute
45.    Next:6
46.    GetEol
47.    GetTokenEx
48.    GetTokenAttribute
49.    Next:11
50.    GetEol
51.    GetTokenEx
52.    GetTokenAttribute
53.    Next:12
54.    GetEol
55.    GetTokenEx
56.    GetTokenAttribute
57.    Next:14
58.    GetEol
59.    GetTokenEx
60.    GetTokenAttribute
61.    Next:15
62.    GetEol
63.    GetTokenEx
64.    GetTokenAttribute
65.    Next:17
66.    GetEol
67.    GetTokenEx
68.    GetTokenAttribute
69.    Next:18
70.    GetEol
71.    GetTokenEx
72.    GetTokenAttribute
73.    Next:24
74.    GetEol
75.    GetTokenEx
76.    GetTokenAttribute
77.    Next:25
78.    GetEol
```

Se puede notar que el editor realiza primero una exploración previa, de toda la línea, antes de aplicar los atributos.

Adicionalmente al coloreado de sintaxis, SynEdit hace su propia exploración independiente para la detección y resaltado de "brackets", (paréntesis, corchetes, llaves y comillas), cuando el cursor se encuentra apuntando a alguno de estos elementos:

```
texto |(texto entre paréntesis (otro texto)) más texto.
```

Sin embargo "SynEdit", permite al resaltador, colaborar con la identificación de estos delimitadores. ¿Por qué?, porque el resaltador puede proporcionar información adicional para el resaltado de los "brackets", ya que maneja los atributos de las diferentes partes del texto.

Para servir a la funcionalidad de "brackets" de SynEdit, el resaltador, debe implementar correctamente, los métodos: "GetToken", y "GetTokenPos" y "GetTokenKind".

¿Cómo funcionan? Para que un "bracket" de apertura, se asocie con su correspondiente "bracket" de cierre se verifica que "tokenKind" devuelve el mismo valor ambos. Si en la exploración se encontrara un "bracket" con un atributo diferente, este no se tomará en cuenta.

Si bien estos métodos no se usan para el coloreado de sintaxis, si determinan el comportamiento del resaltado de "brackets".

Estos métodos son llamados de forma menos frecuente, que los métodos de coloreado de sintaxis. Solo se ejecutan cuando el cursor apunta a un "bracket" o cuando se agrega o quita alguno.

Si ha entendido el proceso de coloreado de sintaxis, ya estamos listos para dar los primeros pasos en la implementación de un resaltador por código.

2.3.3 Primeros pasos

Ante todo es recomendable crear una Unidad especial para almacenar el código de nuestro nuevo resaltador.

Para este ejemplo crearemos una unidad llamada "uSintax", e incluiremos las unidades necesarias para la creación de los objetos a usar.

En esta nueva unidad debemos crear necesariamente una clase derivada de "TSynCustomHighlighter" (definida en la unidad "SynEditHighlighter"), para la creación de nuestro resaltador:

```
{
 Unidad mínima que demuestra la estructura de una clase sencilla que será
 usada para el resaltado de sintaxis. No es funcional, es solo demostrativa.
                                       Creada por Tito Hinostroza: 04/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter;
type
  {Clase para la creación de un resaltador}
  TSynMiColor = class(TSynCustomHighlighter)
  protected
    posIni, posFin: Integer;
    linAct: String;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
```

```
  end;

implementation

constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Es llamado por el editor, cada vez que necesita actualizar la información de
 coloreado sobre una línea. Despues de llamar a esta función, se espera que
 GetTokenEx, devuelva el token actual. Y también después de cada llamada a
 "Next".}
begin
  inherited;
  linAct := NewValue;   //copia la línea actual
  posFin := 1;          //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
{Es llamado por SynEdit, para acceder al siguiente Token. Y es ejecutado por
 cada token de la línea en curso. En este ejemplo siempre se movera un
 caracter.}
begin
  posIni := posFin;     //apunta al siguiente token
  If posIni > length(linAct) then   //¿Fin de línea?
    exit          //salir
  else
    inc(posFin);  //mueve un caracter
end;

function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
  Result := posIni > length(linAct);
end;

procedure  TSynMiColor.GetTokenEx(out  TokenStart:  PChar;  out  TokenLength:
integer);
{Devuelve información sobre el token actual}
begin
  TokenStart := @linAct[posIni];
  TokenLength := posFin - posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
{Devuelve información sobre el token actual}
```

```
begin
  Result := nil;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;

function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;

function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

end.
```

Esta es, probablemente, la clase más simple que se puede implementar para un resaltado de sintaxis. Sin embargo, esta clase no resaltará ningún texto porque no contiene instrucciones para cambiar los atributos del texto. Se limita simplemente a devolver los valores por defecto a las solicitudes de “SynEdit”. No tiene utilidad, es simplemente un ejemplo minimalista de demostración.

Los métodos que aparecen como “override” son los que se requieren implementar para darle la funcionalidad de coloreado, a nuestro resaltador.

Con cada llamada a “SetLine”, se guarda una copia de la cadena en “linAct”, luego se utiliza esta cadena para ir extrayendo los tokens.

A cada petición de “Next”, esta unidad solo devuelve el siguiente carácter que se encuentra en la línea y el atributo devuelto por “GetTokenAttribute”, es siempre NIL, que significa que no hay atributos.

Los métodos “GetToken”, “GetTokenPos” y “GetTokenKind”, tampoco devuelven valores significativos, sino los valores nulos correspondientes.

La clase de resaltador que hemos creado, se llama “TSynMiColor”. No es posible usar la misma clase “TSynCustomHighlighter” como resaltador, porque esta clase es abstracta, y solo se usa para canalizar apropiadamente los requerimientos de TsynEdit, al realizar el coloreado de sintaxis.

Para usar la nueva sintaxis, debemos crear un objeto y asociarla al componente TSynEdit que vayamos a usar. Si tenemos nuestro formulario principal en Unit1 y nuestro objeto TsynEdit se llama “editor”, entonces el código para el uso de esta sintaxis podría ser:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses ... uSintax;
...
var
  Sintaxis  : TSynMiColor;
...
procedure TForm1.FormCreate(Sender: TObject);
...
  Sintaxis := TSynMiColor.Create(Self);  //crea resaltador
  editor.Highlighter := Sintaxis;        //asigna la sintaxis al editor
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  editor.Highlighter := nil;  //quita resaltador
  Sintaxis.Destroy;      //libera objeto
end;
```

Entender el funcionamiento básico de este esquema de trabajo, es el primer paso para la creación de resaltadores funcionales. Si no ha entendido el funcionamiento de este ejemplo sencillo, le recomiendo que repase el código antes de pasar a las siguientes secciones.

2.3.4 Agregando funcionalidad a la sintaxis

El ejemplo anterior, se creo solo con fines didácticos. No se cumple con la funcionalidad deseada, pero se muestra la estructura que debe tener toda clase de coloreado de sintaxis.

Para comenzar, debemos tener en mente que los métodos a implementar, deben ser de ejecución rápida. No deben estar cargados de mucho procesamiento, porque son llamados repetidamente para cada línea modificada del editor, así que no admiten retrasos, de otra forma el editor se volverá pesado y lento.

La primera modificación que debemos introducir, es el método de almacenamiento de la cadena. Cuando se llama a “SetLine”, se debe tener información, sobre la cadena. Pero la clase padre “TSynCustomHighlighter”, guarda ya una copia de la cadena, antes de llamar a “SetLine”.

Por ello no es eficiente crear una copia nueva para nosotros. Bastará con guardar una referencia, un puntero a esta cadena, almacenada en “TSynCustomHighlighter”.

Esto implica modificar la variable “linAct”, para que sea un “PChar”, en lugar de un string. Esto se hace en la definición de la clase. Los métodos “SetLine”, “Next”, “GetEol”, “GetTokenEx” y “GetTokenAttribute”, también deben cambiar:

Nuestro esqueleto de unidad quedaría:

```
{
```

```
 Unidad mínima que demuestra la estructura de una clase sencilla que será
 usada para el resaltado de sintaxis. No es funcional, es solo demostrativa.
                                     Creada por Tito Hinostroza: 04/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter;
type
  {Clase para la creación de un resaltador}
  TSynMiColor = class(TSynCustomHighlighter)
  protected
    posIni, posFin: Integer;
    linAct: PChar;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Es llamado por el editor, cada vez que necesita actualizar la información de
 coloreado sobre una línea. Despues de llamar a esta función, se espera que
 GetTokenEx, devuelva el token actual. Y también después de cada llamada a
 "Next".}
begin
  inherited;
  linAct := PChar(NewValue);  //copia la línea actual
  posFin := 0;                //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
{Es llamado por SynEdit, para acceder al siguiente Token. Y es ejecutado por
```

```
 cada token de la línea en curso. En este ejemplo siempre se movera un
 caracter.}
begin
  posIni := posFin;    //apunta al siguiente token
  if linAct[posIni] = #0 then exit;   ///¿apunta al final?
  inc(posFin);  //mueve un caracter
end;

function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
  Result := linAct[posIni] = #0;
end;

procedure TSynMiColor.GetTokenEx(out TokenStart: PChar; out TokenLength:
integer);
{Devuelve información sobre el token actual}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
{Devuelve información sobre el token actual}
begin
  Result := nil;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;

function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;

function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

end.
```

Ahora vemos que debemos iniciar “posFin” a cero, que es donde empieza ahora la cadena, en “linAct”.

Pero aún esta clase está vacía de atributos. Lo primero que deberíamos hacer es crear nuestros atributos. Estos se deben declarar como propiedades del objeto “TSynMiColor”:

```
    fAtriComent  : TSynHighlighterAttributes;
    fAtriIdent   : TSynHighlighterAttributes;
    fAtriClave   : TSynHighlighterAttributes;
    fAtriNumero  : TSynHighlighterAttributes;
    fAtriEspac   : TSynHighlighterAttributes;
    fAtriCadena  : TSynHighlighterAttributes;
```

Todos los atributos son de tipo “TSynHighlighterAttributes”. Esta clase contiene los diversos atributos que se le pueden asociar a un token, como color de texto, color de fondo, color de borde, etc.

Luego en el constructor, debemos crear y definir las propiedades de estos atributos:

```
constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
  //atributo de comentarios
  fAtriComent  := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic];     //en cursiva
  fAtriComent.Foreground := clGray;    //color de letra gris
  AddAttribute(fAtriComent);
  //atribuuto de palabras claves
  fAtriClave   := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold];        //en negrita
  fAtriClave.Foreground:=clGreen;      //color de letra verde
  AddAttribute(fAtriClave);
  //atributo de números
  fAtriNumero  := TSynHighlighterAttributes.Create('Number');
  fAtriNumero.Foreground := clFuchsia;
  AddAttribute(fAtriNumero);
  //atributo de espacios. Sin atributos
  fAtriEspac   := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);
  //atributo de cadenas
  fAtriCadena  := TSynHighlighterAttributes.Create('String');
  fAtriCadena.Foreground := clBlue;   //color de letra azul
  AddAttribute(fAtriCadena);
end;
```

Se han definido atributos de varias categorías de tokens. Aquí es donde se define la apariencia que tendrá el texto de los tokens.

Para crear un atributo, se recomienda usar las constantes pre-definidas en la unidad “SynEditStrConst”:

```
  SYNS_AttrASP                  =  'Asp';
  SYNS_AttrAssembler            =  'Assembler';
```

```
  SYNS_AttrAttributeName        = 'Attribute Name';
  SYNS_AttrAttributeValue       = 'Attribute Value';
  SYNS_AttrBlock                = 'Block';
  SYNS_AttrBrackets             = 'Brackets';
  SYNS_AttrCDATASection         = 'CDATA Section';
  SYNS_AttrCharacter            = 'Character';
  SYNS_AttrClass                = 'Class';
  SYNS_AttrComment              = 'Comment';
  SYNS_AttrIDEDirective         = 'IDE Directive';
  SYNS_AttrCondition            = 'Condition';
  SYNS_AttrDataType             = 'Data type';
  SYNS_AttrDefaultPackage       = 'Default packages';
  SYNS_AttrDir                  = 'Direction';
  SYNS_AttrDirective            = 'Directive';
  SYNS_AttrDOCTYPESection       = 'DOCTYPE Section';
...
```

La forma de crear un atributo, usando estas constantes, sería identificar primero el tipo de atributo que vamos a crear y elegir la constante que mejor la describa. Para la mayoría de sintaxis, estas serían:

```
SYNS_AttrComment
SYNS_AttrReservedWord
SYNS_AttrNumber
SYNS_AttrSpace
SYNS_AttrString
SYNS_AttrSymbol
SYNS_AttrDirective
SYNS_AttrAssembler
```

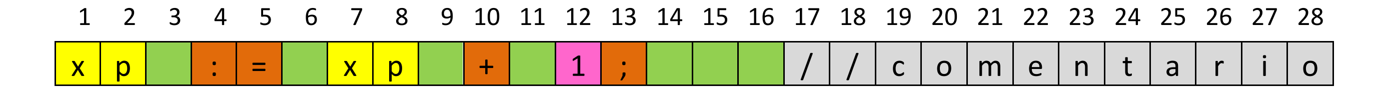
Por lo tanto, para crear un atributo para palabras claves, podríamos que hacer:

```
  fAtriClave     :=     TSynHighlighterAttributes.Create(SYNS_AttrReservedWord,
SYNS_XML_AttrReservedWord);
```

Usar constantes predefinidas, para crear los atributos, no es obligatorio, ni necesario para el funcionamiento del resaltador, pero es una buena práctica si deseamos que nuestros resaltadores, puedan trabajar correctamente con otros programas de Lazarus. Para más información sobre atributos, ver la sección 2.4.3 - Atributos.

Debemos recordar que todos los elementos de la línea a explorar, debe ser necesariamente un token, inclusive los espacios y símbolos.

El siguiente ejemplo, muestra cómo se puede dividir una cadena en tokens diversos:



En este ejemplo el primer token, se define por los caracteres 1 y 2, y se muestra en amarillo. El segundo token es un espacio en blanco y se indica con el color verde. Los caracteres 4 y 5 pueden considerarse como un solo token o como dos tokens distintos. El carácter 12 es un token que seguramente estará en la categoría de números. Los caracteres 14, 15 y 16 se deben agrupar en un solo token espacio de 3 caracteres de ancho (sería ineficiente tratarlo como 3 tokens). A partir del carácter 17, se encuentra un token que abarca hasta el fin de la línea.

Los límites del token, lo define el resaltador (que funciona como extractor de tokens o “lexer”). El editor hará caso, sumisamente, a lo indicado por este objeto, coloreándolo de acuerdo a los atributos entregados.

Como se ve en el ejemplo, todos los caracteres de la línea deben pertenecer a un token. El tamaño de un token va desde un carácter hasta el total de caracteres de la línea. El editor procesará más rápidamente la línea, mientras menos tokens hayan en ella.

Para identificar fácilmente a los atributos, es conveniente crear una enumeración para los atributos de tokens:

```
  //ID para categorizar a los tokens
  TtkTokenKind = (tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString,
tkUnknown);
```

El identificador “tkUnknown”, indicará que el token actual no se ha identificado. En este caso, se asumirá que no tiene atributos.

Y necesitamos, además, un campo para identificar al token actual:

```
TSynMiColor = class(TSynCustomHighlighter)
   ...
   fTokenID: TtkTokenKind;  //Id del token actual
   ...
end;
```

Ahora cuando queramos asignar un atributo, al token actual, debemos poner en “fTokenID”, el identificador del token.

Una vez creados los atributos, debemos agregar funcionalidad al método “Next”, para que pueda extraer los tokens adecuadamente, de la línea de trabajo. La implementación debe ser lo más eficiente posible, por ello usaremos el método de tabla de funciones o de métodos.

La idea es leer el carácter de un token, y de acuerdo a su valor ASCCI, llamamos a una función apropiada, para tratar ese carácter. Para que la llamada sea eficiente, creamos una tabla y la llenamos con punteros a las funciones adecuadas.

```
Type
  TProcTableProc = procedure of object; //Tipo procedimiento para procesar el
                                        //token por el carácter inicial.
...

TSynMiColor = class(TSynCustomHighlighter)
```

```
  protected
    ...
    fProcTable: array[#0..#255] of TProcTableProc; //tabla de funciones
    ...
end;
```

El tipo “TProcTableProc” es un método simple que define procedimientos sin parámetros (así la llamada se hace más rápida). Este tipo de procedimiento es el que se llamará cuando se identifique el carácter inicial de algún token.

Ahora que se tiene definido el tipo de procedimiento a usar, se debe crear estos procedimientos de tratamientos de tokens y llenar la tabla de métodos con sus direcciones. El siguiente código es un ejemplo sencillo de llenado de la tabla de métodos:

```
...

procedure TSynMiColor.CreaTablaDeMetodos;
{Construye la tabla de las funciones usadas para cada caracter inicial del
 tóken a procesar. Proporciona una forma rápida de procesar un token por el
 caracter inicial}
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      '_','A'..'Z','a'..'z': fProcTable[I] := @ProcIdent;
      #0  : fProcTable[I] := @ProcNull; //Caracter de marca de fin de cadena
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
end;
```

Este método, hace corresponder la dirección de una función a cada una de las 256 de las posiciones de la tabla “fProcTable[]”.

El procedimiento “ProcIdent”, es la que se llama cuando se detecta un carácter alfabético (o guion), porque corresponde al inicio de un identificador. Su implementación es sencilla:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  fTokenID := tkKey;
end;
```

La cadena "linAct", se va explorando hasta encontrar un carácter que no sea un carácter válido para un identificador. Observar que no se considera los caracteres del código ASCII extendido (á,é,í, etc). En este ejemplo sencillo, no se distingue el tipo de identificador, sino que se le asigna a todos el atributo "tkKey". Si se quisiera elegir, solo a algunas palabras para marcarlas como "tkKey", se debe hacer aquí.

El procedimiento "ProcNull", se llama al detectar el carácter NUL, es decir el fin de la cadena. Así que su procesamiento solo reduce a marcar "fTokenID" como "tkNull".

```
procedure TSynMiColor.ProcNull;
//Procesa la ocurrencia del caracter #0
begin
  fTokenID := tkNull;   //Solo necesita esto para indicar que se llegó al
final de la línea
end;
```

Observar que no se avanza más en la exploración de la cadena. Este procedimiento es importante para detectar el final de la cadena y permite implementar "GetEol", de forma sencilla:

```
function TSynMiColor.GetEol: Boolean;
//Indica cuando se ha llegado al final de la línea
begin
  Result := fTokenId = tkNull;
end;
```

El procedimiento "ProcSpace", permite procesar los bloques de espacios en blanco. Para los fines de sintaxis, se considerará espacios en blanco, los primeros 32 caracteres del código ASCII, exceptuando los caracteres #10 y #13 que corresponden a saltos de línea:

```
procedure TSynMiColor.ProcSpace;
//Procesa caracter que es inicio de espacio
begin
  fTokenID := tkSpace;
  repeat
    Inc(posFin);
  until (linAct[posFin] > #32) or (linAct[posFin] in [#0, #10, #13]);
end;
```

El carácter de tabulación #9 y el espacio #32, están considerados como espacios en blanco. A estos caracteres en blanco, se le asigna el atributo "tkSpace", que normalmente debe estar sin resaltado.

El otro procedimiento importante, es "ProcUnknown", que está destinado a procesar a todos aquellos tokens que no están considerados dentro de una categoría especial. En nuestro caso será todos los símbolos y números:

```
procedure TSynMiColor.ProcUnknown;
begin
  inc(posFin);
  while (linAct[posFin] in [#128..#191]) OR // continued utf8 subcode
```

```
    ((linAct[posFin]<>#0) and (fProcTable[linAct[posFin]] = @ProcUnknown)) do
inc(posFin);
  fTokenID := tkUnknown;
end;
```

Es importante tener siempre un procedimiento de este tipo para considerar todos aquellos tokens que no son categorizados en grupos predefinidos. Observar que también se consideran los caracteres UTF-8 del código ASCII extendido. Esto es normal ya que SynEdIt trabaja solamente con UTF-8.

Una vez definidos estos procedimientos básicos, se debe implementar la llamada en el método "Next". El código tendría la siguiente forma:

```
procedure TSynMiColor.Next;
//Es llamado por SynEdit, para acceder al siguiente Token.
begin
  posIni := posFin;            //apunta al siguiente token
  fProcTable[linAct[posFin]]; //Se ejecuta la función que corresponda.
end;
```

Aunque no resulta obvio, se puede apreciar la llamada a la función de procesamiento apropiada para cada carácter. Obviamente se debe haber llenado primeramente "fProcTable".

Este modo de procesamiento resulta bastante rápido si se compara con un conjunto de condicionales o hasta con una sentencia "case .. of".

La función de procesamiento asignada, se encargará de actualizar el índice "posFin", que debe quedar siempre apuntando al inicio del siguiente token o al fin de la cadena.

Para que la sintaxis sea reconocida, solo falta modificar "GetTokenAttribute", para indicarle al editor, que atributo debe usar para cada token:

```
function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Devuelve información sobre el token actual
begin
  case fTokenID of
    tkComment: Result := fAtriComent;
    tkKey     : Result := fAtriClave;
    tkNumber  : Result := fAtriNumero;
    tkSpace   : Result := fAtriEspac;
    tkString  : Result := fAtriCadena;
    else Result := nil;  //tkUnknown, tkNull
  end;
end;
```

Tal como hemos definido nuestro resaltador, se reconocerán todas las palabras como palabras claves, y se mostrarán en color verde. Los símbolos y demás caracteres imprimibles, se mostrarán sin atributos, es decir que tomarán el color por defecto del texto.

La siguiente pantalla muestra como quedaría un texto simple, usando este resaltador:

Form1

```
1  numero := numero + 1; //prueba
```

El código completo de la unidad quedaría así:

```
{
 Unidad mínima que demuestra la estructura de una clase sencilla que será usada
 para el resaltado de sintaxis. No es funcional, es solo demostrativa.
                                         Creada por Tito Hinostroza: 04/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter;
type
  {Clase para la creación de un resaltador}

  //ID para categorizar a los tokens
  TtkTokenKind = (tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString, tkUnknown);

  TProcTableProc = procedure of object; //Tipo procedimiento para procesar el
                                        //token por el carácter inicial.
  { TSynMiColor }
  TSynMiColor = class(TSynCustomHighlighter)
  protected
    posIni, posFin: Integer;
    linAct    : PChar;
    fProcTable: array[#0..#255] of TProcTableProc;   //tabla de procedimientos
    fTokenID : TtkTokenKind;   //Id del token actual
    //define las categorías de los "tokens"
    fAtriComent  : TSynHighlighterAttributes;
    fAtriClave   : TSynHighlighterAttributes;
    fAtriNumero  : TSynHighlighterAttributes;
    fAtriEspac   : TSynHighlighterAttributes;
    fAtriCadena  : TSynHighlighterAttributes;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
  private
    procedure CreaTablaDeMetodos;
```

```
    procedure ProcIdent;
    procedure ProcNull;
    procedure ProcSpace;
    procedure ProcUnknown;
  end;

implementation

constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
  //atributo de comentarios
  fAtriComent  := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic];     //en cursiva
  fAtriComent.Foreground := clGray;    //color de letra gris
  AddAttribute(fAtriComent);
  //atribuuto de palabras claves
  fAtriClave   := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold];       //en negrita
  fAtriClave.Foreground:=clGreen;     //color de letra verde
  AddAttribute(fAtriClave);
  //atributo de números
  fAtriNumero  := TSynHighlighterAttributes.Create('Number');
  fAtriNumero.Foreground := clFuchsia;
  AddAttribute(fAtriNumero);
  //atributo de espacios. Sin atributos
  fAtriEspac   := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);
  //atributo de cadenas
  fAtriCadena  := TSynHighlighterAttributes.Create('String');
  fAtriCadena.Foreground := clBlue;   //color de letra azul
  AddAttribute(fAtriCadena);

  CreaTablaDeMetodos;  //Construte tabla de métodos
end;

procedure TSynMiColor.CreaTablaDeMetodos;
{Construye la tabla de las funciones usadas para cada caracter inicial del tóken a
procesar.
 Proporciona una forma rápida de procesar un token por el caracter inicial}
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      '_','A'..'Z','a'..'z': fProcTable[I] := @ProcIdent;
      #0      : fProcTable[I] := @ProcNull;   //Se lee el caracter de marca de fin de
cadena
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
end;

procedure TSynMiColor.ProcIdent;
```

```
//Procesa un identificador o palabra clave
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  fTokenID := tkKey;
end;

procedure TSynMiColor.ProcNull;
//Procesa la ocurrencia del caracter #0
begin
  fTokenID := tkNull;   //Solo necesita esto para indicar que se llegó al final de la
línae
end;
procedure TSynMiColor.ProcSpace;
//Procesa caracter que es inicio de espacio
begin
  fTokenID := tkSpace;
  repeat
    Inc(posFin);
  until (linAct[posFin] > #32) or (linAct[posFin] in [#0, #10, #13]);
end;
procedure TSynMiColor.ProcUnknown;
begin
  inc(posFin);
  while (linAct[posFin] in [#128..#191]) OR  //continued utf8 subcode
   ((linAct[posFin]<>#0)   and   (fProcTable[linAct[posFin]]   =   @ProcUnknown))   do
inc(posFin);
  fTokenID := tkUnknown;
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Es llamado por el editor, cada vez que necesita actualizar la información de
 coloreado sobre una línea. Despues de llamar a esta función, se espera que
 GetTokenEx, devuelva el token actual. Y también después de cada llamada a
 "Next".}
begin
  inherited;
  linAct := PChar(NewValue);  //copia la línea actual
  posFin := 0;                //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
//Es llamado por SynEdit, para acceder al siguiente Token.
begin
  posIni := posFin;           //apunta al siguiente token
  fProcTable[linAct[posFin]]; //Se ejecuta la función que corresponda.
end;

function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
  Result := fTokenId = tkNull;
end;
```

```
procedure TSynMiColor.GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
{Devuelve información sobre el token actual}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Devuelve información sobre el token actual
begin
  case fTokenID of
    tkComment: Result := fAtriComent;
    tkKey     : Result := fAtriClave;
    tkNumber : Result := fAtriNumero;
    tkSpace   : Result := fAtriEspac;
    tkString : Result := fAtriCadena;
    else Result := nil;  //tkUnknown, tkNull
  end;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;

function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;

function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

end.
```

2.3.5 Propiedad GetDefaultAttribute

Puede que alguien se haya preguntado ¿Cómo acceder, desde fuera de la clase, a los atributos de, por ejemplo, las palabras claves? Recordemos que los atributos de los tokens se deben declarar en el resaltador y no en la clase padre “TSynCustomHighlighter”.

Una respuesta sencilla, sería “ponemos las propiedades de atributo como públicos, y luego podremos referenciarlo como cualquier, propiedad de nuestro resaltador.

Y es cierto, eso funcionaría, pero si la pregunta fuera: ¿Cómo acceder desde el editor a los atributos de los tokens? Entonces ahí si se complica un poco la situación, porque, a pesar de que el editor (de la clase TSyenEdit) tiene la propiedad “HighLighter”, esta solo hace referencia a la clase

"TSynCustomHighlighter" y no a la clase derivada (resaltador) que siempre usamos para implementar el coloreado.

Para solventar, en parte, esta dificultad, existe un método adicional que es recomendable implementar. Este método es "GetDefaultAttribute" y permitirá a nuestro resaltador, responder a las peticiones de acceso a los atributos que genere "TSynCustomHighlighter".

A pesar de que la clase "TSynCustomHighlighter" no incluye propiedades de tipo atributo (se deja al programador la libertad de crear los que desee), si incluye una forma de acceder a los atributos principales de los tokens. En la clase se han definido las propiedades fijas:

```
property CommentAttribute: TSynHighlighterAttributes;
property IdentifierAttribute: TSynHighlighterAttributes;
property KeywordAttribute: TSynHighlighterAttributes;
property StringAttribute: TSynHighlighterAttributes;
property SymbolAttribute: TSynHighlighterAttributes;
property WhitespaceAttribute: TSynHighlighterAttributes;
```

Que permiten leer o modificar los atributos indicados. Sin embargo, para que estas propiedades funcionen, nosotros debemos sobre-escribir (override) en nuestro resaltador, el siguiente método:

```
function TSynMiColor.GetDefaultAttribute(Index: integer):
TSynHighlighterAttributes;
{Este método es llamado por la clase "TSynCustomHighlighter", cuando se
accede a alguna de
 sus propiedades: CommentAttribute, IdentifierAttribute, KeywordAttribute,
StringAttribute,
 SymbolAttribute o WhitespaceAttribute.
}
begin
  case Index of
    SYN_ATTR_COMMENT   : Result := fCommentAttri;
    SYN_ATTR_IDENTIFIER: Result := fIdentifierAttri;
    SYN_ATTR_KEYWORD   : Result := fKeyAttri;
    SYN_ATTR_WHITESPACE: Result := fSpaceAttri;
    SYN_ATTR_STRING    : Result := fStringAttri;
    else Result := nil;
  end;
end;
```

Como se ve, la idea es darle acceso a nuestros atributos, de acuerdo al tipo de atributo, solicitado. Desde luego, si no hemos definido un atributo específico, podríamos devolver NIL. De la misma forma, es posible que hayamos definido atributos adicionales que podrían no ser accesibles desde fuera de la clase, porque no se encuentran en la categoría solicitada.

Cuando alguien accede a la propiedad "CommentAttribute", de "TSynCustomHighlighter", esta llama a "GetDefaultAttribute", pasando el parámetro "SYN_ATTR_COMMENT". Es decisión del programador, devolver el atributo que considere necesario. Lo común sería devolver el atributo que

represente a los comentarios, pero la clase no hará ninguna validación posterior. En teoría podríamos devolver el atributo que deseemos.

Si no va a haber accesos a las propiedades mencionadas de “TSynCustomHighlighter”, no es necesario implementar “GetDefaultAttribute”, sin embargo, es recomendable implementarlo siempre.

2.3.6 Reconociendo palabras claves.

En el ejemplo anterior marcamos a todos los identificadores como palabras claves asignándole el atributo “tkKey”. Esto lo hacíamos en el método “ProcIdent”:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  fTokenID := tkKey;
end;
```

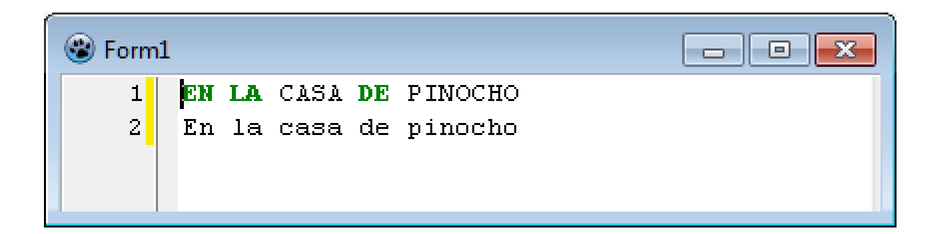
Pero en un caso normal, solo se marcarán algunos identificadores como palabras claves. Para ello, el camino más sencillo podría ser, comparar el token actual, con un grupo de palabras claves, y solo en caso de que coincidan, marcarlas como palabras claves.

El código a usar, podría ser como este:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
var tam: integer;
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  tam := posFin - posIni;
  if strlcomp(linAct + posIni,'EN',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'DE',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LA',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LOS',tam) = 0 then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
```

La comparación de cadenas se hace usando la función “strlcomp”, porque estamos manejando una variable “PChar”.

En este código se reconocen solo las palabras “EN”, “LA” Y “DE” como palabras reservadas. Al aplicar esta modificación podríamos tener una pantalla como esta.



Observar que las rutinas solo reconocen las mayúsculas, porque la comparación de cadenas se hace de esta forma.

Para ir agregando más palabras, se puede ir aumentando la lista y elegir los atributos dadas a cada categoría de palabras. Sin embargo, este método se vuelve pesado, conforme crece la cantidad de palabras y condicionales a agregar y por lo tanto no es el camino ideal a seguir en la implementación de una sintaxis adecuada.

Más adelante veremos como podemos optimizar la detección de identificadores.

2.3.7 Optimizando la sintaxis.

Como hemos estado insistiendo a lo largo de esta descripción, es importante mantener un código rápido en aras del buen desempeño del editor. Para ello debemos identificar los puntos donde podamos reducir ciclos de ejecución.

Analizando el código anterior se puede ver que el procedimiento "ProcIdent", es el más pesado en cuanto a procesamiento. Por su implementación requiere hacer múltiples comparaciones y verificaciones para detectar los identificadores a colorear.

La primera optimización que haremos tiene que ver con la condición:

```
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
```

A pesar, de que el uso de conjuntos resulta eficiente, este código puede optimizarse considerablemente si se usa una tabla de selección.

Poder identificar rápidamente si un carácter se encuentra en una lista, es fácil si enfocamos el problema desde otra perspectiva. Imaginemos que cada carácter está asociado a una tabla que contiene como valor una bandera simple que indica si la variable es o no es parte de la lista. Una estructura así sería del tipo:

```
  Identifiers: array[#0..#255] of ByteBool;
```

Ahora creamos un procedimiento de llenado que marque solo las casillas de caracteres válidos para identificadores, como "true":

```
procedure CreaTablaIdentif;
var
  i: Char;
begin
```

```
  for I := #0 to #255 do
  begin
    Case i of
      '_', '0'..'9', 'a'..'z', 'A'..'Z': Identifiers[i] := True;
    else
      Identifiers[i] := False;
    end;
  end;
end;
```

Una vez llenada esta tabla, ya podemos usarla para detectar rápidamente, que caracteres se consideran como parte de un identificador, usando nuevamente un simple while:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
var tam: integer;
begin
  while Identifiers[linAct[posFin]] do Inc(posFin);
  tam := posFin - posIni;
  if strlcomp(linAct + posIni,'EN',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'DE',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LA',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LOS',tam) = 0 then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
```

El siguiente punto a optimizar Está en las comparaciones múltiples. Lógicamente en este ejemplo, solo hay 4 comparaciones, pero normalmente podemos estar trabajando con más de 100. En estas condiciones, aunque no lo parezca, se puede estar perdiendo un tiempo valioso en detección de cadenas, haciendo muchas veces, verificaciones redundantes.

El problema se reduce en optimizar la comparación de una cadena en una lista de varias. Existen diversos métodos para llevar a cabo la optimización de esta tarea.

La mayoría de los componentes de sintaxis de Lazarus, usan el método de las funciones Hash (Hash-functions) que es un poco complejo, pero que básicamente se trata en asignarle a cada palabra clave, a detectar, un valor numérico, más o menos único, que permita categorizarlo en un número pequeño de grupos (Ver Apéndice para más detalle sobre este método).

Aunque este método es rápido, no es legible y confunde fácilmente. Además las modificaciones sencillas, como agregar una nueva palabra clave, requiere de un cálculo cuidadoso antes de modificar el código.

Aquí usaremos un método que es generalmente más rápido y mucho más legible y fácil de modificar. Este algoritmo es una forma simplificada de un árbol de prefijos. Se puede ver como un árbol en el que se implementa solo el primer nivel. Como prefijo se usa el primer carácter del identificador a buscar. A este método lo llamaremos el algoritmo del Primer carácter como prefijo.

El método se implementa creando una primera categorización de las palabras usando la misma tabla de métodos creada en “CreaTablaDeMetodos”, creando una función para cada letra inicial del identificador. Así el código de “CreaTablaDeMetodos”, tendría la siguiente forma:

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of

      ...

      'A','a': fProcTable[I] := @ProcA;
      'B','b': fProcTable[I] := @ProcB;
      'C','c': fProcTable[I] := @ProcC;
      'D','d': fProcTable[I] := @ProcD;
      'E','e': fProcTable[I] := @ProcE;
      'F','f': fProcTable[I] := @ProcF;
      'G','g': fProcTable[I] := @ProcG;
      'H','h': fProcTable[I] := @ProcH;

      ...
      'H','h': fProcTable[I] := @ProcH;

end;
```

Luego en los procedimientos ProcA, ProcB, ... etc, se realiza el procesamiento de un grupo reducido de identificadores, reduciendo sensiblemente la cantidad de comparaciones.

Por ejemplo, el procedimiento encargado de identificar las palabras claves, empezadas en “L”, sería:

```
procedure TSynMiColor.ProcL;
var tam: integer;
begin
  while Identifiers[linAct[posFin]] do Inc(posFin);
  tam := posFin - posIni;
  if strlcomp(linAct + posIni,'LA',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LOS',tam) = 0 then fTokenID := tkKey else
    fTokenID := tkUnknown;  //sin atributos
end;
```

Se nota, que se reduce considerablemente la cantidad de comparaciones a realizar. De hecho, si las palabras clave a comparar, estuvieran distribuidas uniformemente en el alfabeto, la cantidad de comparaciones se reduciría en 26 veces.

Tal como está este procedimiento, solo detectará las palabras reservadas en mayúscula[7]. Para hacerlo insensible a la caja, se debería agregar un procesamiento adicional.

Aprovechamos esta funcionalidad faltante para optimizar las comparaciones, usando una función de comparación rápida que, además ignore la caja (mayúscula o minúscula). Para ello usaremos nuevamente la invaluable ayuda de las tablas. EL método consistirá en crear una tabla que asigne un ordinal a cada carácter alfabético, independientemente de su caja. A esta tabla la llamaremos “mHashTable”, y aprovecharemos para llenarla en “CreaTablaIdentif”:

```
procedure CreaTablaIdentif;
var
  i, j: Char;
begin
  for i := #0 to #255 do
  begin
    Case i of
      '_', '0'..'9', 'a'..'z', 'A'..'Z': Identifiers[i] := True;
    else Identifiers[i] := False;
    end;
    j := UpCase(i);
    Case i in ['_', 'A'..'Z', 'a'..'z'] of
      True: mHashTable[i] := Ord(j) - 64
    else
      mHashTable[i] := 0;
    end;
  end;
end;
```

Ahora con esta función creada, ya podemos crear una función, para comparaciones rápidas. Usaremos la que se usan en las librerías de Lazarus:

```
function TSynMiColor.KeyComp(const aKey: String): Boolean;
//Compara rápidamente una cadena con el token actual, apuntado por "fToIden".
//El tamaño del token debe estar en "fStringLen"
var
  I: Integer;
  Temp: PChar;
begin
  Temp := fToIdent;
  if Length(aKey) = fStringLen then
  begin
    Result := True;
    for i := 1 to fStringLen do
    begin
      if mHashTable[Temp^] <> mHashTable[aKey[i]] then
      begin
```

[7] También se puede ver que el reconocimiento de palabras no es efectivo porque reconocerá las palabras aunque solo coincidan en los primeros caracteres.

```
        Result := False;
        break;
      end;
      inc(Temp);
    end;
  end else Result := False;
end;
```

Esta función de comparación usa el puntero “fToIdent” y la variable “fStringLen”, para evaluar la comparación. El único parámetro que requiere es la cadena a comparar.

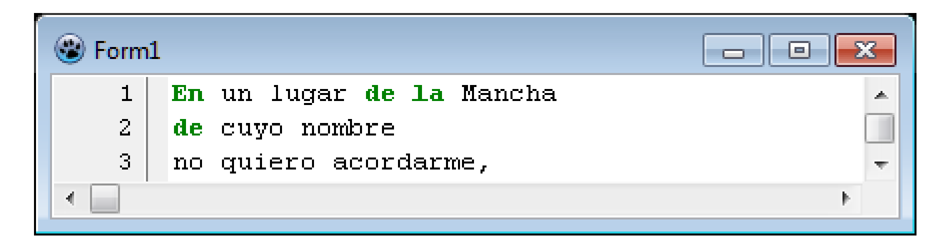
Con la ayuda de la tabla “mHashTable”, la comparación se hará ignorando la caja.

Ahora podemos replantear el procedimiento “ProcL”:

```
procedure TSynMiColor.ProcL;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('A')     then fTokenID := tkKey else
  if KeyComp('OS')     then fTokenID := tkKey else
    fTokenID := tkUnknown;  //sin atributos
end;
```

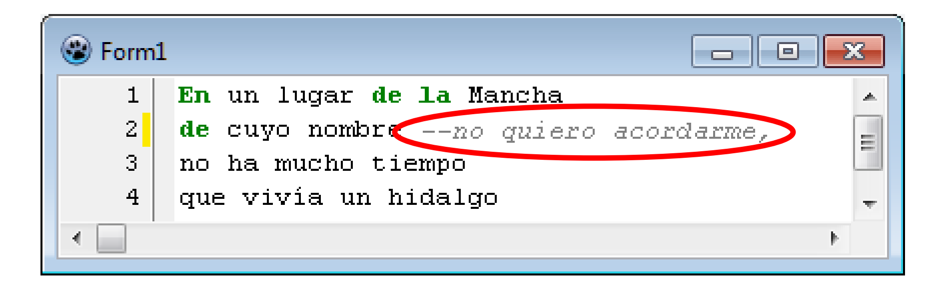
Este procedimiento verifica si se detectan las palabras “LA” o “LOS”. Como una optimización adicional, se omite la comparación del primer carácter, ya que este se ha detectado antes de llamar a esta función.

Ahora ya tenemos lista nuestro resaltador básico. Una prueba del programa con unas palabras claves más, nos dará el siguiente resultado:



2.3.8 Coloreado de comentarios de una sola línea

Se debe hacer en el resaltador. El procedimiento es sencillo. Se debe detectar primero la secuencia de caracteres, que corresponden al inicio del comentario, y luego ubicar el fin de la línea.



En nuestro ejemplo agregamos a nuestra función “MakeMethodTables”, la detección de comentarios, identificando el carácter guion “-”, ya que el token para comentarios de una línea es el doble guion “--”.

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
...
      '-'     : fProcTable[I] := @ProcMinus;
...
    end;
end;
```

Solo podemos detectar un carácter en “`CreaTablaDeMetodos`”, así que es necesario detectar el siguiente carácter en la función “ProcMinus”.

Esta función debe tener la siguiente forma:

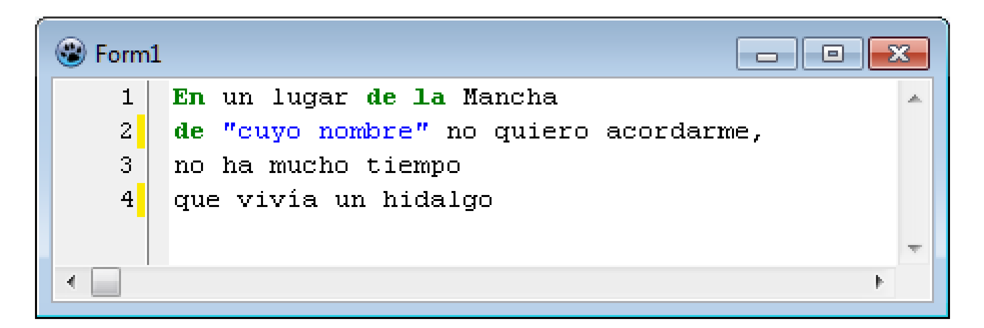
```
procedure TSynMiColor.ProcMinus;
//Procesa el símbolo '-'
begin
  case LinAct[PosFin + 1] of   //ve siguiente caracter
    '-':                     //es comentario de una sola línea
      begin
        fTokenID := tkComment;
        inc(PosFin, 2);        //salta a siguiente token
        while not (linAct[PosFin] in [#0, #10, #13]) do Inc(PosFin);
      end;
  else  //debe ser el operador "menos".
    begin
      inc(PosFin);
      fTokenID := tkUnknown;
    end;
  end
end;
```

Es necesario ver el siguiente carácter, para determinar si se trata del token de comentario. De ser así, se busca el final de línea, para considerar todo ese bloque como un solo token con atributo “tkComent”.

De no ser el token de comentario, simplemente pasamos al siguiente token, marcando el símbolo "-" como de tipo "tkUnknown". Si deseamos considerar al signo "-" en una categoría especial, este es el punto donde debe hacerse.

2.3.9 Coloreado de cadenas

El siguiente caso de coloreado, corresponde al coloreado de cadenas. Este es un caso simple, porque las cadenas ocuparán a lo más una línea. Una cadena tiene un delimitador, que se usa tanto para el inicio como para el fin de la cadena.



En nuestro caso, usaremos las cadenas delimitadas por comillas.

Primero debemos incluir su detección en el procedimiento "CreaTablaDeMetodos":

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
...
      '"'     : fProcTable[I] := @ProcString;
...
    end;
end;
```

Al detectarse el carácter comilla, se pasará el control a "ProcString", quien se encargará de buscar el delimitador final de la cadena, y marcar toda la cadena como un solo token:

```
procedure TSynMiColor.ProcString;
//Procesa el caracter comilla.
begin
  fTokenID := tkString;   //marca como cadena
  Inc(PosFin);
  while (not (linAct[PosFin] in [#0, #10, #13])) do begin
    if linAct[PosFin] = '"' then begin //busca fin de cadena
      Inc(PosFin);
      if (linAct[PosFin] <> '"') then break;  //si no es doble comilla
    end;
    Inc(PosFin);
  end;
```

```
end;
```

Observar que antes de determinar si se ha encontrado el final de la cadena, se verifica primero que no se trate de un caso de doble comilla. Normalmente una doble comilla "entre comillas" representa al carácter comillas.

Se puede deducir también que el token de cadena termina necesariamente en la misma línea donde empezó. Es posible generar coloreado de cadenas de múltiples líneas, como si fueran comentarios multi-líneas, que es el caso que veremos a continuación.

2.3.10 Manejo de rangos

Antes de ver como se implementa el coloreado de rangos, es conveniente conocer el manejo de rangos, en el editor SynEdit.

El uso de rangos permite colorear elementos que pudieran extenderse hasta más allá de una línea. Tal es el caso de los comentarios o cadenas de múltiples líneas que implementan muchos lenguajes.

Para implementar esta funcionalidad, el editor maneja tres métodos que están definidos en la clase "TSynCustomHighlighter" de la unidad "SynEditHighlighter":

```
  TSynCustomHighlighter = class(TComponent)
   ...
    function GetRange: Pointer; virtual;
    procedure SetRange(Value: Pointer); virtual;
    procedure ResetRange; virtual;
   ...
   end;

function TSynCustomHighlighter.GetRange: pointer;
begin
  Result := nil;
end;

procedure TSynCustomHighlighter.SetRange(Value: Pointer);
begin
end;

procedure TSynCustomHighlighter.ResetRange;
begin
end;
```

Estos métodos tienen la siguiente función:

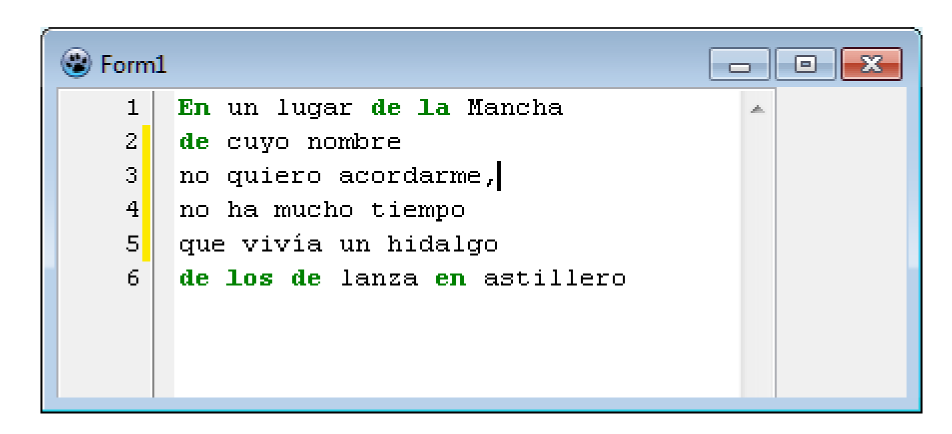
- ResetRange.- Se ejecuta antes de explorar la primera línea del texto, ya que no hay líneas anteriores que afecten el rango.

- GetRange.- Es llamado después de terminar la exploración de una línea, para obtener el nivel al final dela línea. Este valor es almacenado internamente.
- SetRange.- Es llamado antes de la exploración de una línea. Cambia el nivel usando el nivel de la línea anterior.

La secuencia en la ejecución, depende de la acción realizada. La primera vez que se abre un documento, SynEdit explora todas las líneas con GetRange() (leyendo todos los tokens), para tener el valor que le corresponde a cada línea. El valor leído, quedará almacenado internamente, para las comparaciones posteriores.

Cada vez que se modifica una parte del documento, SynEdit hace diversas llamadas a GetRange() y SetRange(), para reconstruir el nuevo estado del documento. La exploración, puede ir desde la línea actual, hasta el final del documento, si SynEdit lo juzga necesario. Sin embargo, lo común es que la exploración del documento solo se haga en unas pocas líneas.

El siguiente ejemplo muestra un editor, y las llamadas a los métodos de rangos y "SetLine", cuando se modifica la línea 3:




```
1.   SetRange
2. SetLine: no quiero acordarme,
3.   GetRange
4. SetLine: no ha mucho tiempo
5.   GetRange
6. SetLine: que vivía un hidalgo
7.   GetRange
8.   SetRange
9. SetLine: de cuyo nombre
10.  SetRange
11.SetLine: no quiero acordarme,
12.  SetRange
13.SetLine: no ha mucho tiempo
14.  SetRange
15.SetLine: que vivía un hidalgo
```

El editor suele explorar el texto desde una línea antes de la línea modificada, hasta encontrar que una línea devuelve el mismo nivel que tenía anteriormente.

Si no se va a realizar tratamiento de sintaxis, no es necesario sobrescribir estos métodos. Solo deben modificarse, cuando se va a implementar coloreado por rangos, o plegado de código.

El valor que envía "SetRange", en el parámetro "Value", es un puntero, así como el valor que espera recibir "GetRange", porque han sido diseñados para trabajar con objetos. Pero no es necesario trabajar con punteros. En la práctica se suele usar un tipo enumerado para identificar los niveles de los rangos, teniendo cuidado de hacer las conversiones necesarias.

```
Type
   ...
   TRangeState = (rsUnknown, rsComment);
   ...

TSynMiColor = class(TSynCustomHighlighter)
   ...
   fRange: TRangeState;
   ...
end;
   ...
function TSynMiColor.GetRange: Pointer;
begin
  Result := Pointer(PtrInt(fRange));
end;

procedure TSynMiColor.SetRange(Value: Pointer);
begin
  fRange := TRangeState(PtrUInt(Value));
end;
```

Las funciones PtrInt y PtrUInt, convierten un puntero a un entero del mismo tamaño que el puntero.

El valor de los punteros no es importante en sí[8], porque no se hace acceso a los objetos apuntados por ellos, en condiciones normales. Lo importante es que tomen valores diferentes cuando se encuentra un rango particular (comentarios, bloques, etc.), de cuando no hay rangos activos.

En el siguiente ejemplo, se ha implementado el coloreado de comentarios (/* … */), y se muestran los valores de los parámetros pasados y leídos, cuando se inserta un comentario a partir de la línea 3:

[8] En el diseño de "TSynCustomHighlighter", se ha definido el uso de punteros, con el fin de poder usar referencias a objetos reales, que puedan asociarse a un rango específico. Para la mayoría de casos, nos bastará con manejar un simple entero o enumerado. Sin embargo, en ciertos desarrollos, como la clase "TSynCustomFoldHighlighter", si se hace uso de objetos para su funcionalidad. El uso de punteros para manejar rangos, resulta confuso al principio, pues queda la sensación de que hubiera bastado con usar un simple entero, pero el diseño actual permite mayor libertad.

Form1

```
En un lugar de la Mancha
de cuyo 'nombre'
/* no quiero acordarme,
no quiero acordarme,
no ha mucho tiempo*/
que vivia un hidalgo
de los de lanza en astillero
```

```
1.   SetRange: 0
2. SetLine: /*no quiero acordarme,
3.   GetRange: 1
4. SetLine: no quiero acordarme,
5.   GetRange: 1
6. SetLine: no ha mucho tiempo*/
7.   GetRange: 0
8.   SetRange: 0
9. SetLine: de cuyo 'nombre'
10.  SetRange: 0
11.SetLine: /*no quiero acordarme,
12.  SetRange: 1
13.SetLine: no quiero acordarme,
14.  SetRange: 1
15.SetLine: no ha mucho tiempo*/
```

Al lado de “SetRange” o “GetRange”, se está mostrando el ordinal de “fRange”, como una ayuda visual para ver cómo va cambiando. Aclaramos que el cambio en “fRange” no tiene por qué ser consecutivo, basta con que “fRange” tome valores diferentes, para que la funcionalidad de coloreado, trabaje.

El concepto de rangos, puede tornarse un poco difícil de asimilar, al principio. Puede ayudar pensar en ellos como una forma de guardar información adicional que le corresponda a cada línea. Por lo tanto existe un valor de rango (puntero a un objeto), por cada línea del texto explorado.

El rango no está asociado al concepto de “nivel”[9], o “alcance”, es simplemente: Información adicional asociada a cada línea que se puede usar para guardar el estado de una línea, al final de ella. Desde luego que si no se quiere usar esta información, basta con dejarla con los valores por defecto.

Puede haber mucha información adicional, que se quiera asociar a cada línea. Los rangos son una forma útil de guardar esta información. El editor lee al inicio (a través del resaltador), el rango de cada línea, al terminar de explorar la línea y guarda esta información.

Luego cuando se modifica el texto, el editor hace exploraciones sucesivas, para “actualizar” los rangos en las líneas que pueden verse afectadas.

[9] Es posible que se le pueda dar ese significado, en una implementación particular, pero no es obligatorio.

Por norma, el rango de las líneas anteriores no se afecta por el cambio en una línea cualquiera. Pero las líneas sucesivas si pueden alterarse en su rango. Por ello, de ser el caso, el editor explorará las líneas siguientes hasta que encuentre que, el rango que le corresponde es similar al que tenía, y cesa la actualización.

Si es que la información que queremos asociar, no se refiere a una línea, sino a elementos más pequeños, los rangos no ayudarán directamente[10]. Son más útiles, cuando la información a almacenar, se puede obtener directa o fácilmente a partir del estado de la línea anterior.

En el manejo de rangos, debe cumplirse que:

“Conocer el estado final de la línea anterior (rango u objeto apuntado por el rango), es todo lo que se necesita para trabajar correctamente con la línea actual”

Otra forma de ver la utilidad de los rangos, es pensar en ellos como una ayuda para trasladar información de una línea a otra, considerando que el editor no explora ordenadamente todo el texto[11], sino que trata de hacer la menor cantidad de exploraciones, realizando varias exploraciones pequeñas en diversas partes del texto, de acuerdo al texto modificado.

Esto significa que si alteramos por ejemplo, la variable XXX, mientras se explora la línea “n”, no podemos estar seguros de que el valor de XXX, se leerá al explorar la línea “n+1”, porque no necesariamente la siguiente línea a explorar será la “n+1”. Si quisiéramos que al explorarse la línea “n+1”, se vea el valor deseado de la variable XXX, se debe ver la forma de guardarla y recuperarla como parte del rango.

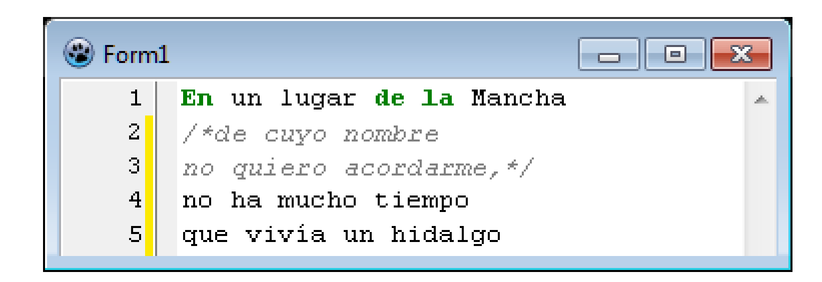
La siguiente sección ayudará en la comprensión de rangos, al ver una implementación real en el coloreado de sintaxis.

2.3.11 Coloreado de rango o contexto

El coloreado de contexto, involucra considerar un intervalo del texto, que puede estar en una o varias líneas, como si fuera un único token. Lógicamente, por la jerarquía usada, un token, no puede ser mayor a una línea, por ello, si el rango se extiende a más de una línea, se identificará como varios tokens (uno por cada línea) de la misma categoría.

[10] Aunque podría idearse formas que permitan reconstruir la información del interior de la línea, en un objeto rango.

[11] Solo dentro de una línea, los tokens son explorados, siempre de forma secuencial hasta el final de la línea.



Dada nuestra clase, este coloreado se puede hacer usando una clase derivada del resaltador usado, o se puede incluir la funcionalidad en el mismo código de la sintaxis original.

Si se desea crear una clase derivada, debe tener una estructura parecida a esta:

```
  TSynDemoHlContextFoldBase = class(TSynMiColor)
  protected
    FCurRange: Integer;
  public
    procedure Next; override;
    function GetTokenAttribute: TSynHighlighterAttributes; override;
  public
```

En los métodos "Next" y "GetTokenAttribute", se debe agregar el comportamiento adicional que se necesita para colorear los rangos de texto.

Para mayor información, se recomienda ver el ejemplo que viene con Lazarus: en \examples\SynEdit\NewHighlighterTutorial\

Sin embargo, la forma más eficiente, sería incluir esta funcionalidad en el mismo resaltador.

Ahora vamos a considerar el caso de colorear un rango de una o varias líneas. Para este ejemplo consideremos el coloreado de comentarios de múltiples líneas.

Primero elegimos los caracteres delimitadores de comentarios. Para nuestro ejemplo usaremos los típicos caracteres de C: "/*" y "*/". Todo el texto que se encuentre entre estos caracteres será considerado como un comentario y tendrá el atributo "fStringAttri".

El delimitador de inicio debemos detectarlo por el carácter "/", pero haciendo la validación, ya que podría tratarse del operador de división.

Nuevamente hacemos la interceptación en la función "`CreaTablaDeMetodos`":

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
...
      '/'     : fProcTable[I] := @ProcSlash;
...
```

```
    end;
end;
```

Y la identificación de comentarios la hacemos en “ProcSlash”:

```
procedure TSynMiColor.ProcSlash;
//Procesa el símbolo '/'
begin
  case linAct[PosFin + 1] of
    '*':            //comentario multi-línea
      begin
        fRange := rsComment;      //marca rango
        inc(PosFin, 2);
        CommentProc;   //Procesa en modo comentario
      end;
  else      //debe ser el operador "entre".
    begin
      inc(PosFin);
      fTokenID := tkUnknown;
    end;
  end
end;
```

Observamos que estamos trabajando con un procedimiento “CommentProc” y con una nueva bandera, llamada “fRange”. Que se debe declarar como se muestra:

```
Type
   ...
   TRangeState = (rsUnknown, rsComment);
   ...

TSynMiColor = class(TSynCustomHighlighter)
   ...
   fRange: TRangeState;
   ...
```

Esta declaración es importante para el manejo de rangos. La detección de coloreado en rangos, requiere este tipo de manejo.

Se pueden crear diversos tipos de rangos en “TRangeState”, de acuerdo a las necesidades de la sintaxis. El orden los enumerados no es importante.

Adicionalmente para implementar la funcionalidad de rangos, se debe sobre-escribir los tres métodos de rangos:

```
TSynMiColor = class(TSynCustomHighlighter)
   ...
```

```
    function GetRange: Pointer; override;
    procedure SetRange(Value: Pointer); override;
    procedure ResetRange; override;
    ...
end;

    ...

{Implementación de las funcionalidades de rango}
procedure TSynMiColor.ReSetRange;
begin
  fRange := rsUnknown;
end;

function TSynMiColor.GetRange: Pointer;
begin
  Result := Pointer(PtrInt(fRange));
end;

procedure TSynMiColor.SetRange(Value: Pointer);
begin
  fRange := TRangeState(PtrUInt(Value));
end;
```

Una vez definido el comportamiento de estos métodos. El editor se encargará de la gestión de sus llamadas, ahorrándonos el trabajo de controlar el estado de las líneas.

Pero hace falta aún, procesar las líneas que estén en el rango de comentarios. Para ello, implementamos el método “CommentProc”:

```
procedure TSynMiColor.CommentProc;
begin
  fTokenID := tkComment;
  case linAct[PosFin] of
    #0:
      begin
        ProcNull;
        exit;
      end;
  end;
  while linAct[PosFin] <> #0 do
    case linAct[PosFin] of
      '*':
        if linAct[PosFin + 1] = '/' then
        begin
          inc(PosFin, 2);
          fRange := rsUnknown;
          break;
        end
        else inc(PosFin);
```

```
      #10: break;
      #13: break;
    else inc(PosFin);
    end;
end;
```

Este método, explora las líneas en busca del delimitador final del comentario. Si no lo encuentra considera todo lo explorado (inclusive la línea completa) como un solo token de tipo “tkComment”. Hay que notar que no se está detectando el delimitador de fin de comentario “*/” en ninguna otra parte de la clase.

Que debe ser llamado cuando se detecte que estamos en medio de un comentario, como se hace en “ProcSlash”, pero debemos también incluirlo en “Next”:

```
procedure TSynMiColor.Next;
begin
  posIni := PosFin;   //apunta al primer elemento
  if fRange = rsComment then
     CommentProc
  else
    begin
      fRange := rsUnknown;
      fProcTable[linAct[PosFin]]; //Se ejecuta la función que corresponda.
    end;
end;
```

De esta forma, pasamos el control a “CommentProc”, cuando nos encontremos en medio de un comentario de contexto.

2.4 La clase TSynCustomHighlighter

Hasta ahora, se ha mostrado como implementar un resaltador simple, usando solamente las propiedades y métodos necesarios de TSynCustomHighlighter. Ahora vamos a hablar un poco de la clase en sí y de algunas funcionalidades adicionales que trae.

Una rápida mirada al código de la clase, nos indicará que es una clase más o menos extensa, para ser una clase abstracta. Pero, a pesar de todo, la clase TSynCustomHighlighter, no almacena información en sí. Toda la información que maneja, se guarda en el editor, específicamente en Lines[].

Todo editor que desee implementar el resaltado de sintaxis, debe estar asociado a un resaltador. Las relaciones entre un editor y un resaltador son:

- Un editor puede estar asociado a uno y solo un resaltador.
- Un resaltador puede servir a uno o varios editores.

Esta relación, se puede deducir, tomando en consideración que el resaltador en sí, no almacena información del texto que explora.

Una consecuencia, de las relaciones anteriores, es que un resaltador no está asociado, de manera fija, a un editor en particular.

Pero en condiciones normales, cuando un editor utiliza un solo resaltador, la relación simplificada es de uno a uno.

Una característica saltantes de TSynCustomHighlighter, es la forma como explora las líneas usando el concepto de rango.

La información de rangos, que genera el resaltador, la guarda en el editor y la usa para acceder a cada línea, con el estado inicial apropiado.

2.4.1 CurrentLines[] y CurrentRanges[]

Una de las propiedades más útiles de TSynCustomHighlighter es CurrentLines[]. Este arreglo, permite acceder al "buffer" del editor actual, es decir, el que esté haciendo uso del resaltador en ese momento.

El arreglo CurrentLines[], es asignado a Lines[] antes que el editor use el resaltador, así que siempre se garantizará que CurrentLines[], hace referencia al editor actual.

De la misma forma a como CurrentLines[], nos permite acceder a la información de las líneas actuales, la propiedad CurrentRanges[][12], nos permite acceder a los valores de rango, que le corresponde a cada línea del texto.

[12] Hay que aclarar que CurrentRanges[], se encuentra declarado como PROTECTED, así que no es directamente accesible desde fuera de la clase, pero se puede hacer accesible al momento de crear nuestro propio resaltador.

CurrentRanges[], es un objeto complejo, pero para fines prácticos, lo podemos ver como una simple tabla de punteros. Estos valores, son los que se asignan a cada línea, cuando TSynCustomHighlighter, lo solicita al resaltador mediante GetRange().

Es necesario saber que CurrentLines[] y CurrentRanges[], son como tablas que empiezan en cero. Si por ejemplo quisiéramos ver el valor de rango de la línea 3, tendríamos que ver en CurrentRanges[2].

Lógicamente CurrentLines[]y CurrentRanges[] tienen el mismo tamaño en todo momento, y este tamaño coincide con la cantidad de líneas que tiene el editor actual (cuando el resaltador está asignado a alguno).

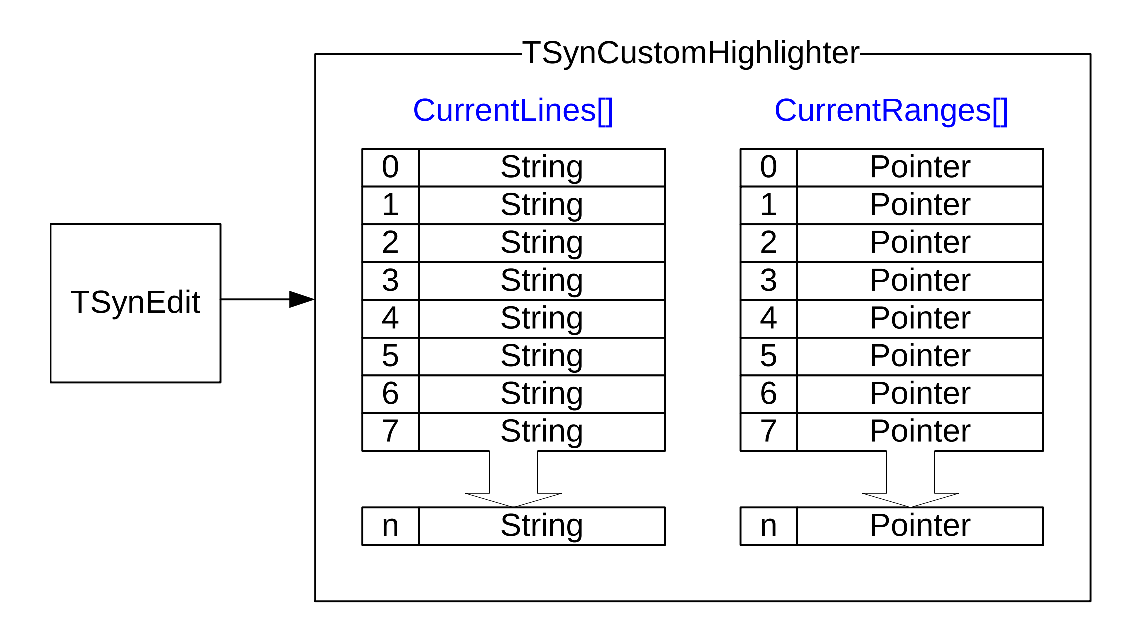
Intentar acceder a CurrentLines[] o CurrentRanges[], cuando el resaltador no está asignado a un editor, generará un error en tiempo de ejecución.

La tabla CurrentLines[], devuelve cadenas, pero CurrentRanges[], devuelve punteros. Devuelve los mismos punteros que son asignados cuando se usan los métodos:

- TSynCustomHighlighter.GetRange()
- TSynCustomHighlighter.SetRange()

Es decir que CurrentRanges[] nos permite recuperar la información del Rango almacenada en cada línea .

El siguiente esquema aclarará lo dicho:



Desde luego que el significado que tenga este puntero, es el mismo que se usa en el resaltador (enumerado, entero, referencia a objeto, etc). Corresponderá al programador, aplicar la conversión de tipos, correspondiente a cada caso.

2.4.2 Algunos Métodos y Propiedades

La clase TSynCustomHighlighter tiene, adicionalmente, diversas propiedades útiles, que nos pueden servir en algún momento.

El método ScanAllRanges(), hace que el editor explore de nuevo, todas las líneas para actualizar las referencias al rango que tiene cada línea. Este método es útil, por ejemplo, cuando se cambia la sintaxis del lenguaje de trabajo.

La propiedad "LanguageName", devuelve una cadena con un texto correspondiente al nombre del lenguaje para el que está preparado el resaltador. Es responsabilidad del resaltador activar esta propiedad, sobrescribiendo el método GetLanguageName().

La propiedad "SampleSource", devuelve una cadena con un trozo de código de ejemplo, en el lenguaje usado, como muestra para probar al resaltador. Nuevamente, es responsabilidad del resaltador activar esta propiedad, sobrescribiendo el método GetLanguageName().

Existe una propiedad interna que puede usarse para el trabajo con identificadores: IdentChars. Internamente, es una simple referencia a GetIdentChars(), que tiene la siguiente definición:

```
function TSynCustomHighlighter.GetIdentChars: TSynIdentChars;
begin
  Result := [#33..#255];
end;
```

Puede que sea necesario, sobrescribir este método para que se adapte a nuestras necesidades.

El método StartAtLineIndex(), permite posicionar el resaltador en una línea cualquiera del texto, para iniciar la exploración de esa línea. Esto es útil cundo queremos realizar una exploración adicional (fuera del proceso normal de exploración del resaltador), para obtener alguna información adicional del resaltador.

Un ejemplo de uso de esta función, se puede observar en el método: TSynEdit. GetHighlighterAttriAtRowCol(), del editor, que permite obtener el atributo y el token para una posición cualquiera del texto.

2.4.3 Atributos

Gran parte del código de TSynCustomHighlighter, está referido al manejo de atributos. Estos permiten configurar como se verán los tokens en el editor.

Los atributos son objetos que se deberían crear antes de usar el resaltador. Para crear un atributo nuevo, se ejecuta:

```
  fKeyAttri          :=          TSynHighlighterAttributes.Create(SYNS_AttrKey,
SYNS_XML_AttrKey);
  fKeyAttri.Style := [fsBold];
  AddAttribute(fKeyAttri);  //guarda la referencia al atributo
```

Por lo general, este código se coloca siempre en el constructor del resaltador. Pero podrían crearse dinámicamente en cualquier parte del proceso.

Las referencias a los atributos se guardan en una estructura interna de la clase (fAttributes). Como facilidad adicional, se liberan al destruir la clase, así que no es necesario (ni se debe), destruir los atributos agregados al resaltador.

Todos los resaltadores, tienen definidos (de forma estática o dinámica) varios atributos, porque son necesarios para asignar las propiedades de resaltado del texto.

Los atributos son objetos de la clase TSynHighlighterAttributes, definida en la misma unidad SynEditHighlighter, que contiene a la clase TSynCustomHighlighter.

La clase TSynCustomHighlighter , no tiene atributos definidos internamente, como es su forma de trabajo, pero tiene definidas propiedades que permiten acceder a algunos de los atributos más comunes:

```
  TSynCustomHighlighter = class(TComponent)
...
  public
    property AttrCount: integer read GetAttribCount;
    property Attribute[idx: integer]: TSynHighlighterAttributes
      read GetAttribute;
    property Capabilities: TSynHighlighterCapabilities
       read {$IFDEF SYN_LAZARUS}FCapabilities{$ELSE}GetCapabilities{$ENDIF};
    property SampleSource: string read GetSampleSource write SetSampleSource;
    property CommentAttribute: TSynHighlighterAttributes
      index SYN_ATTR_COMMENT read GetDefaultAttribute;
    property IdentifierAttribute: TSynHighlighterAttributes
      index SYN_ATTR_IDENTIFIER read GetDefaultAttribute;
    property KeywordAttribute: TSynHighlighterAttributes
      index SYN_ATTR_KEYWORD read GetDefaultAttribute;
    property StringAttribute: TSynHighlighterAttributes
      index SYN_ATTR_STRING read GetDefaultAttribute;
    property             SymbolAttribute:             TSynHighlighterAttributes
//mh 2001-09-13
      index SYN_ATTR_SYMBOL read GetDefaultAttribute;
    property WhitespaceAttribute: TSynHighlighterAttributes
      index SYN_ATTR_WHITESPACE read GetDefaultAttribute;
```

Este acceso depende de que se implemente correctamente el método GetDefaultAttribute(). Para facilidad en la implementación, existen algunas constantes predefinidas en la unidad "SynEditHighlighter":

```
const
  SYN_ATTR_COMMENT           =   0;
  SYN_ATTR_IDENTIFIER        =   1;
  SYN_ATTR_KEYWORD           =   2;
  SYN_ATTR_STRING            =   3;
```

```
  SYN_ATTR_WHITESPACE          =   4;
  SYN_ATTR_SYMBOL              =   5;
```

Estas constantes no cubren a la cantidad de atributos que puede manejar un resaltador medianamente completo, pero son una ayuda sencilla para acceder a los atributos más comunes.

Una implementación típica de GetDefaultAttribute(), en un resaltador es:

```
function TSynLFMSyn.GetDefaultAttribute(Index: integer):
TSynHighlighterAttributes;
begin
  case Index of
    SYN_ATTR_COMMENT: Result := fCommentAttri;
    SYN_ATTR_IDENTIFIER: Result := fIdentifierAttri;
    SYN_ATTR_KEYWORD: Result := fKeyAttri;
    SYN_ATTR_STRING: Result := fStringAttri;
    SYN_ATTR_WHITESPACE: Result := fSpaceAttri;
    SYN_ATTR_SYMBOL: Result := fSymbolAttri;
  else
    Result := nil;
  end;
end;
```

La idea es devolver la referencia apropiada al resaltador, cuando se lo soliciten a través de GetDefaultAttribute(). Los atributos Están definidos en el resaltador.

De lo visto sobre los atributos, se puede deducir que existen diversas formas para acceder a los atributos de un resaltador.

La forma más sencilla para acceder a los atributos, es a través de GetDefaultAttribute():

```
var atributo: TSynHighlighterAttributes;
...
  atributo := SynLFMSyn1.GetDefaultAttribute(SYN_ATTR_KEYWORD);
  atributo.Foreground:=clRed;
```

Sin embargo, esta forma solo funcionará para los atributos que hayan sido correctamente inscritos en el método GetDefaultAttribute(), lo cual depende de la correcta implementación del resaltador. Además solo son visibles algunos atributos más comunes.

Otra forma de acceso a los atributos es usar las propiedades públicas que implementan muchos resaltadores. Estas son:

```
TSynLFMSyn = class(TSynCustomFoldHighlighter)
...
private
  fCommentAttri: TSynHighlighterAttributes;
  fIdentifierAttri: TSynHighlighterAttributes;
```

```
  fKeyAttri: TSynHighlighterAttributes;
  fNumberAttri: TSynHighlighterAttributes;
  fSpaceAttri: TSynHighlighterAttributes;
  fStringAttri: TSynHighlighterAttributes;
  fSymbolAttri: TSynHighlighterAttributes;
...
published
  property CommentAttri: TSynHighlighterAttributes read fCommentAttri
    write fCommentAttri;
  property IdentifierAttri: TSynHighlighterAttributes read fIdentifierAttri
    write fIdentifierAttri;
  property   KeyAttri:   TSynHighlighterAttributes   read   fKeyAttri   write
fKeyAttri;
  property NumberAttri: TSynHighlighterAttributes read fNumberAttri
    write fNumberAttri;
  property SpaceAttri: TSynHighlighterAttributes read fSpaceAttri
    write fSpaceAttri;
  property StringAttri: TSynHighlighterAttributes read fStringAttri
    write fStringAttri;
end;
```

Los atributos en sí están ocultos, pero sus propiedades respectivas están publicadas, así que podrían cambiarse desde el Inspector de Objetos.

Dependiendo de la implementación del resaltador, estas propiedades nos podrían permitir mayor visibilidad para acceder a los atributos del resaltador.

Así una forma de acceder a los atributos, usando estas propiedades sería:

```
var atributo: TSynHighlighterAttributes;
...
  atributo := SynLFMSyn1.KeywordAttribute;
  atributo.Foreground:=clRed;
```

Otra forma de acceder a todos los atributos del resaltador sería usando la tabla de atributos Attribute[], que es una simple referencia al método protegido GetAttribute():

```
function TSynCustomHighlighter.GetAttribute(idx: integer):
  TSynHighlighterAttributes;
begin
  Result := nil;
  if (idx >= 0) and (idx < fAttributes.Count) then
    Result := TSynHighlighterAttributes(fAttributes.Objects[idx]);
end;
```

Los atributos se guardan internamente en fAttributes, un TStringList, en el campo de objetos.

Un acceso típico a un atributo, a través del resaltador sería:

```
var atributo: TSynHighlighterAttributes;
```

```
...
  atributo := SynLFMSyn1.Attribute[2];
  atributo.Foreground:=clRed;
```

La desventaja, es que necesitamos conocer el índice del atributo con el que vamos a trabajar. Como ayuda podríamos usar el nombre del atributo.

Así, una iteración típica de atributos, sería:

```
var i:integer;
begin
...
  for i:=0 to SynLFMSyn1.AttrCount - 1 do
    ShowMessage(SynLFMSyn1.Attribute[i].Name);
...
```

El nombre almacenado en la propiedad "Name", es el nombre que se asigna al atributo, cuando se le crea mediante su constructor:

```
  Atributo := TSynHighlighterAttributes.Create('Nombre');
```

A la vez, podemos usar también el segundo parámetro del constructor (aStoredName):

```
constructor TSynHighlighterAttributes.Create(aCaption: string; aStoredName:
String = '');
```

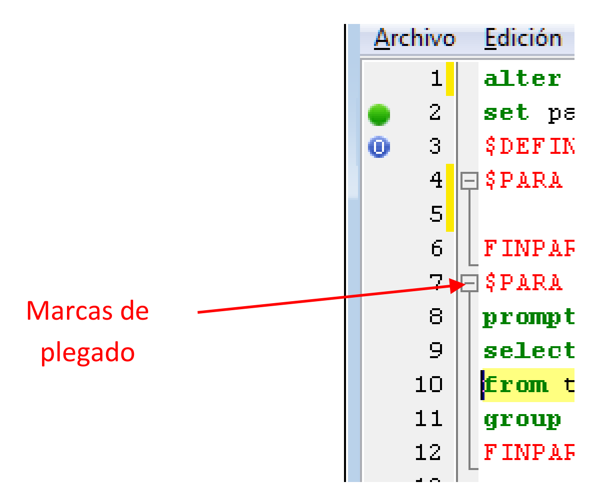
Que permite asignar otro nombre interno al atributo. Luego podremos acceder a este nombre mediante la propiedad "StoredName" del atributo.

Para saber la cantidad de atributos, tenemos la propiedad "AttrCount".

Iterar a través de Attribute[], permite acceder efectivamente a todos los atributos que se hayan creado en el resaltador.

2.5 Funcionalidad de Plegado de Código

La característica de plegado de código o "folding" se refiere a la posibilidad de mostrar la extensión de un bloque de texto predefinido, en la barra izquierda del editor:



La mayoría de los componentes de sintaxis, que vienen en Lazarus, no incluyen la funcionalidad de plegado, así que deberá crearse por código. El código para el plegado, debe estar en el mismo resaltador.

2.5.1 Plegado Básico

Vamos a considerar agregar plegado de código, partiendo de un resaltador sencillo, como el que hemos descrito anteriormente.

Primeramente, para implementar el plegado, se requiere derivar la clase resaltador de la clase "TSynCustomFoldHighlighter" (definida en la unidad "SynEditHighlighterFoldBase"), en lugar de usar la clase "TSynCustomHighlighter".

Esa nueva clase, contiene el código de procesamiento para el plegado.

Primeramente cambiamos la declaración de la clase resaltador que usamos:

```
   TSynMiColor = class(TSynCustomHighlighter)
   ...
```

Por:

```
  TSynDemoHlFold = class(TSynCustomFoldHighlighter)
   ...
```

Pero también debemos modificar los métodos de tratamiento de rangos:

```
{Implementación de las funcionalidades de rango}
procedure TSynMiColor.ReSetRange;
begin
```

```
  inherited;
  fRange := rsUnknown;
end;

function TSynMiColor.GetRange: Pointer;
begin
  CodeFoldRange.RangeType := Pointer(PtrInt(fRange));
  Result := inherited;
end;

procedure TSynMiColor.SetRange(Value: Pointer);
begin
  inherited;
  fRange := TRangeState(PtrUInt(CodeFoldRange.RangeType));
end;
```

Para responder adecuadamente a los requerimientos de la clase. De eso se encarga “inherited”, y movemos la información de contexto a “CodeFoldRange”, porque es ahí donde trabaja ahora “TSynCustomFoldHighlighter”.

Una vez implementadas estas modificaciones, ya estamos listos para agregar el “folding”, para ello debemos detectar, el inicio y final del bloque.

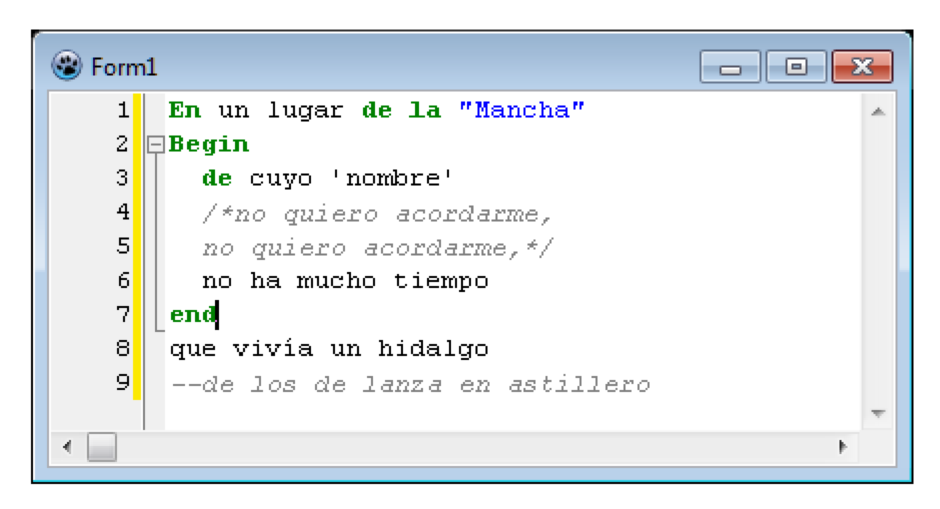
La forma más sencilla es llamando a los métodos: “StartCodeFoldBlock” y “EndCodeFoldBlock”, cuando se detecte el token de inicio y el de fin respectivamente:

Por ejemplo, si agregáramos el inició con la detección de la palabra BEGIN y el fin del bloque con la palabra END, el código sería:

```
procedure TSynMiColor.ProcB;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('EGIN') then begin
      fTokenID := tkKey; StartCodeFoldBlock(nil); end
  else
  if KeyComp('Y')     then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
...
procedure TSynMiColor.ProcE;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('N')  then fTokenID := tkKey else
  if KeyComp('ND') then begin
    fTokenID := tkKey; EndCodeFoldBlock(); end
  else
```

```
    fTokenID := tkUnknown;  //identificador común
end;
```

Un texto de prueba podría mostrarse así:



2.5.2 Código ejemplo

Un código de ejemplo completo con coloreado de sintaxis de identificadores, cadenas, comentarios de una sola línea y multilíneas y "folding", se muestra a continuación:

```
{Unidad mínima funcional que demuestra la estructura de una clase sencilla
 que implementa el coloreado de sintaxis completo y plegado de código.
                                        Por Tito Hinostroza: 07/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter, SynEditHighlighterFoldBase;
type
  {Clase para la creación de un resaltador}
  TRangeState = (rsUnknown, rsComment);
  //ID para categorizar a los tokens
  TtkTokenKind = (tkComment, tkKey, tkNull, tkSpace, tkString, tkUnknown);

  TProcTableProc = procedure of object; //Tipo procedimiento para procesar el
                                        //token por el carácter inicial.
  { TSynMiColor }
  TSynMiColor = class(TSynCustomFoldHighlighter)
  protected
    posIni, posFin: Integer;
    fStringLen: Integer;  //Tamaño del token actual
    fToIdent: PChar;      //Puntero a identificcdor
    linAct   : PChar;
    fProcTable: array[#0..#255] of TProcTableProc; //tabla de procedimientos
    fTokenID : TtkTokenKind;  //Id del token actual
    fRange: TRangeState;
    //define las categorías de los "tokens"
    fAtriComent  : TSynHighlighterAttributes;
```

```
    fAtriClave   : TSynHighlighterAttributes;
    fAtriEspac   : TSynHighlighterAttributes;
    fAtriCadena  : TSynHighlighterAttributes;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
  private
    procedure CommentProc;
    procedure CreaTablaDeMetodos;
    function KeyComp(const aKey: String): Boolean;
    procedure ProcMinus;
    //Funciones de procesamiento de identificadores
    procedure ProcNull;
    procedure ProcSlash;
    procedure ProcSpace;
    procedure ProcString;
    procedure ProcUnknown;

    procedure ProcB;
    procedure ProcC;
    procedure ProcD;
    procedure ProcE;
    procedure ProcL;

    function GetRange: Pointer; override;
    procedure SetRange(Value: Pointer); override;
    procedure ResetRange; override;
  end;

implementation
var
  Identifiers: array[#0..#255] of ByteBool;
  mHashTable: array[#0..#255] of Integer;

procedure CreaTablaIdentif;
var  i, j: Char;
begin
  for i := #0 to #255 do
  begin
    Case i of
      '_', '0'..'9', 'a'..'z', 'A'..'Z': Identifiers[i] := True;
    else Identifiers[i] := False;
    end;
    j := UpCase(i);
    Case i in ['_', 'A'..'Z', 'a'..'z'] of
      True: mHashTable[i] := Ord(j) - 64
```

```
    else
      mHashTable[i] := 0;
    end;
  end;
end;

constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
  //atributo de comentarios
  fAtriComent  := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic];     //en cursiva
  fAtriComent.Foreground := clGray;    //color de letra gris
  AddAttribute(fAtriComent);
  //atribuuto de palabras claves
  fAtriClave   := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold];       //en negrita
  fAtriClave.Foreground:=clGreen;     //color de letra verde
  AddAttribute(fAtriClave);
  //atributo de espacios. Sin atributos
  fAtriEspac   := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);
  //atributo de cadenas
  fAtriCadena  := TSynHighlighterAttributes.Create('String');
  fAtriCadena.Foreground := clBlue;   //color de letra azul
  AddAttribute(fAtriCadena);

  CreaTablaDeMetodos;  //Construye tabla de métodos
end;

procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      '-'    : fProcTable[I] := @ProcMinus;
      '"'    : fProcTable[I] := @ProcString;
      '/'    : fProcTable[I] := @ProcSlash;
      'B','b': fProcTable[I] := @ProcB;
      'C','c': fProcTable[I] := @ProcC;
      'D','d': fProcTable[I] := @ProcD;
      'E','e': fProcTable[I] := @ProcE;
      'L','l': fProcTable[I] := @ProcL;
      #0      : fProcTable[I] := @ProcNull;    //Se lee el caracter de marca de fin de
cadena
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
end;
function TSynMiColor.KeyComp(const aKey: String): Boolean;
var
  I: Integer;
  Temp: PChar;
```

```
begin
  Temp := fToIdent;
  if Length(aKey) = fStringLen then
  begin
    Result := True;
    for i := 1 to fStringLen do
    begin
      if mHashTable[Temp^] <> mHashTable[aKey[i]] then
      begin
        Result := False;
        break;
      end;
      inc(Temp);
    end;
  end else Result := False;
end;

procedure TSynMiColor.ProcMinus;
//Procesa el símbolo '-'
begin
  case LinAct[PosFin + 1] of   //ve siguiente caracter
    '-':                       //es comentario de una sola línea
      begin
        fTokenID := tkComment;
        inc(PosFin, 2);        //salta a siguiente token
        while not (linAct[PosFin] in [#0, #10, #13]) do Inc(PosFin);
      end;
  else  //debe ser el operador "menos".
    begin
      inc(PosFin);
      fTokenID := tkUnknown;
    end;
  end
end;
procedure TSynMiColor.ProcString;
//Procesa el caracter comilla.
begin
  fTokenID := tkString;   //marca como cadena
  Inc(PosFin);
  while (not (linAct[PosFin] in [#0, #10, #13])) do begin
    if linAct[PosFin] = '"' then begin //busca fin de cadena
      Inc(PosFin);
      if (linAct[PosFin] <> '"') then break;  //si no es doble comilla
    end;
    Inc(PosFin);
  end;
end;
procedure TSynMiColor.ProcSlash;
//Procesa el símbolo '/'
begin
  case linAct[PosFin + 1] of
    '*':             //comentario multi-línea
      begin
        fRange := rsComment;     //marca rango
        fTokenID := tkComment;
```

```
        inc(PosFin, 2);
        while linAct[PosFin] <> #0 do
          case linAct[PosFin] of
            '*':
              if linAct[PosFin + 1] = '/' then
              begin
                inc(PosFin, 2);
                fRange := rsUnknown;
                break;
              end else inc(PosFin);
            #10: break;
            #13: break;
          else
            inc(PosFin);
          end;
      end;
  else      //debe ser el operador "entre".
    begin
      inc(PosFin);
      fTokenID := tkUnknown;
    end;
  end
end;

procedure TSynMiColor.ProcB;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('EGIN')  then begin fTokenID := tkKey; StartCodeFoldBlock(nil); end else
  if KeyComp('Y')     then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
procedure TSynMiColor.ProcC;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
    fTokenID := tkUnknown;  //identificador común
end;
procedure TSynMiColor.ProcD;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('E') then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
procedure TSynMiColor.ProcE;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('N')    then fTokenID := tkKey else
  if KeyComp('ND')   then begin fTokenID := tkKey; EndCodeFoldBlock(); end else
```

```
    fTokenID := tkUnknown;  //identificador común
end;
procedure TSynMiColor.ProcL;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('A')    then fTokenID := tkKey else
  if KeyComp('OS')    then fTokenID := tkKey else
    fTokenID := tkUnknown;  //sin atributos
end;

procedure TSynMiColor.ProcNull;
//Procesa la ocurrencia del caracter #0
begin
  fTokenID := tkNull;   //Solo necesita esto para indicar que se llegó al final de la
línae
end;
procedure TSynMiColor.ProcSpace;
//Procesa caracter que es inicio de espacio
begin
  fTokenID := tkSpace;
  repeat
    Inc(posFin);
  until (linAct[posFin] > #32) or (linAct[posFin] in [#0, #10, #13]);
end;
procedure TSynMiColor.ProcUnknown;
begin
  inc(posFin);
  while (linAct[posFin] in [#128..#191]) OR // continued utf8 subcode
   ((linAct[posFin]<>#0)   and   (fProcTable[linAct[posFin]]   =   @ProcUnknown))   do
inc(posFin);
  fTokenID := tkUnknown;
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
begin
  inherited;
  linAct := PChar(NewValue);  //copia la línea actual
  posFin := 0;                //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
begin
  posIni := PosFin;   //apunta al primer elemento
  if fRange = rsComment then
     CommentProc
  else
    begin
      fRange := rsUnknown;
      fProcTable[linAct[PosFin]]; //Se ejecuta la función que corresponda.
    end;
end;
```

```
function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
  Result := fTokenId = tkNull;
end;

procedure TSynMiColor.GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
{Devuelve información sobre el token actual}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Devuelve información sobre el token actual
begin
  case fTokenID of
    tkComment: Result := fAtriComent;
    tkKey    : Result := fAtriClave;
    tkSpace  : Result := fAtriEspac;
    tkString : Result := fAtriCadena;
    else Result := nil;  //tkUnknown, tkNull
  end;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;
function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;
function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

procedure TSynMiColor.CommentProc;
begin
  fTokenID := tkComment;
  case linAct[PosFin] of
    #0:
      begin
        ProcNull;
        exit;
      end;
  end;
  while linAct[PosFin] <> #0 do
    case linAct[PosFin] of
      '*':
        if linAct[PosFin + 1] = '/' then
```

```
        begin
          inc(PosFin, 2);
          fRange := rsUnknown;
          break;
        end
        else inc(PosFin);
      #10: break;
      #13: break;
    else inc(PosFin);
    end;
end;

///////// Implementación de las funcionalidades de rango /////////
procedure TSynMiColor.ReSetRange;
begin
  inherited;
  fRange := rsUnknown;
end;

function TSynMiColor.GetRange: Pointer;
begin
  CodeFoldRange.RangeType := Pointer(PtrInt(fRange));
  Result := inherited;
end;

procedure TSynMiColor.SetRange(Value: Pointer);
begin
  inherited;
  fRange := TRangeState(PtrUInt(CodeFoldRange.RangeType));
end;

initialization
   CreaTablaIdentif;   //Crea la tabla para búsqueda rápida
end.
```

No se han incluido todos los comentarios, por temas de espacio.

2.5.3 Mejorando el Plegado

La forma en que se ha mostrado el plegado, es un proceso bastante sencillo. Sin embargo, el plegado en SynEdit, es bastante completo e incluye funcionalidades adicionales.

Hasta ahora podemos resumir que para controlar el plegado se requiere de los métodos StartCodeFoldBlock() y EndCodeFoldBlock(), que tienen las siguientes declaraciones:

```
function StartCodeFoldBlock(ABlockType: Pointer;
      IncreaseLevel: Boolean = true): TSynCustomCodeFoldBlock; virtual;
procedure EndCodeFoldBlock(DecreaseLevel: Boolean = True); virtual;
```

El funcionamiento de estos métodos es simple. StartCodeFoldBlock() agrega un bloque de plegado y EndCodeFoldBlock() quita el último método agregado.

No hay mayor magia en esto. Cada llamada a EndCodeFoldBlock(), eliminará siempre el último plegado agregado con StartCodeFoldBlock().

En su forma más sencilla, para iniciar un bloque de plegado, usaríamos:

```
  StartCodeFoldBlock(nil);
```

Y para cerrar un bloque de plegado haríamos:

```
  EndCodeFoldBlock();
```

Esta forma de trabajo sería suficiente para un plegado simple. Sin embargo, podríamos necesitar saber cuál es el bloque que estamos manejando para poder decidir si es válido o no, cerrar el bloque actual. Por ejemplo sabemos que en Pascal la palabra reservada UNTIL, cierra el bloque REPEAT, pero no un bloque BEGIN.

Para poder saber qué bloque es el que estamos manejando, podríamos crear una estructura auxiliar, como una cola (ya que los bloques de plegado se pueden anidar), y poder leer siempre cual es el último bloque en el que nos encontramos.

Pero no es necesario tal trabajo, porque dicha estructura ya existe en la clase "TSynCustomFoldHighlighter", y está preparada especialmente para este trabajo.

Para identificar a nuestro bloque de plegado actual, debemos usar el parámetro "ABlockType" de StartCodeFoldBlock().

Para ello, podría resultar conveniente tener una lista de identificadores de bloques de plegado en nuestro resaltador:

```
  TMisTipoDeBloques = (
    cfbt_BeginEnd,
```

```
    cfbt_RepeatUntil,
    cfbt_RecordEnd,
    cfbt_uses,
    cfbt_var
    );
```

Luego cuando queramos indicar el tipo de bloque, usaríamos el siguiente código:

```
  StartCodeFoldBlock(Pointer(PtrInt(ABlockType)));
```

La conversión de tipo es necesaria, porque el parámetro “ABlockType”, es de tipo puntero.

Cerrar este bloque no ofrece ninguna diferencia a cualquier otro bloque. Bastaría una simple llamada a EndCodeFoldBlock():

```
  EndCodeFoldBlock();
```

No es necesaria mayor información, porque se sabe que siempre será el último bloque, el que se cerrará.

Para saber, cuál es el último bloque en la pila de plegados, contamos con el método TopCodeFoldBlockType():

```
    function TopCodeFoldBlockType(DownIndex: Integer = 0): Pointer;
```

El parámetro es opcional, y por lo general debe quedarse en cero, a menos que se desee obtener otro bloque de plegado más interior.

Ahora sí, con los métodos: StartCodeFoldBlock(), EndCodeFoldBlock() y TopCodeFoldBlockType(), tenemos todo lo necesario para poder crear estructuras complejas de plegado dentro de un resaltador.

Ahora se pueden realizar comparaciones de tipo:

```
var
  p: Pointer;
...
  p := TopCodeFoldBlockType;  //lee último bloque ingresado
  //verifica si corresponde cerrar el bloque actual
  if TMisTipoDeBloques(PtrUInt(p)) in [cfbt_BeginEnd, cfbt_RecordEnd] then
    EndCodeFoldBlock(DecreaseLevel);
...
```

2.5.4 Más sobre el Plegado

En los métodos StartCodeFoldBlock() y EndCodeFoldBlock(), existe un parámetro que permite manejar la visibilidad de la marca de plegado. Este es "IncreaseLevel" para StartCodeFoldBlock() y "DecreaseLevel" para EndCodeFoldBlock().

Llamar a StartCodeFoldBlock(), con "IncreaseLevel" en FALSE, no mostrará la típica marca de plegado en la parte izquierda del editor (Gutter). Pero sin embargo, el plegado se realizará internamente, o mejor dicho, se generará información interna de la misma forma a cómo se haría con un plegado visible, pero sin que se permita plegar el código.

Esto puede resultar confuso. Alguien podría preguntar ¿Por qué haría tal cosa? ¿Por qué crearía un plegado que no es visible? ¿Qué utilidad tendría un bloque de plegado que no se pueda plegar?

La respuesta sencilla sería: Porque de esta forma se permite habilitar o deshabilitar el plegado sin alterar la estructura de un texto.

Si manejamos diversos bloques de plegado, y hay relaciones entre estos bloques, no es conveniente quitar alguno de ellos porque se podría alterar la estructura de los demás bloques de plegado. Por ello, cuando se desea "eliminar" algún bloque de plegado, lo que se hace en realidad es simplemente ocultarlo, poniendo "IncreaseLevel" a FALSE, al momento de llamar a StartCodeFoldBlock().

De la misma forma, para ser consistente, se debe poner siempre "DecreaseLevel" a FALSE, en EndCodeFoldBlock(), cuando se ha creado el bloque con "IncreaseLevel" en FALSE.

Conozcamos un poco mejor a los métodos principales. El método StartCodeFoldBlock(), tiene la siguiente implementación:

```
function TSynCustomFoldHighlighter.StartCodeFoldBlock(ABlockType: Pointer;
  IncreaseLevel: Boolean = True): TSynCustomCodeFoldBlock;
begin
  Result:=CodeFoldRange.Add(ABlockType, IncreaseLevel);
end;
```

El método EndCodeFoldBlock(), tiene la siguiente implementación:

```
procedure TSynCustomFoldHighlighter.EndCodeFoldBlock(DecreaseLevel: Boolean =
True);
begin
  CodeFoldRange.Pop(DecreaseLevel);
end;
```

El objeto "CodeFoldRange", funciona como una pila a la que se le van agregando (Add) y quitando la información de plegado.

La propiedad "CodeFoldRange", es una referencia a "FCodeFoldRange", de la clase TSynCustomHighlighterRange:

```
TSynCustomHighlighterRange = class
private
  FCodeFoldStackSize: integer; // EndLevel
  FMinimumCodeFoldBlockLevel: integer;
  FRangeType: Pointer;
  FTop: TSynCustomCodeFoldBlock;
public
  constructor Create(Template: TSynCustomHighlighterRange); virtual;
  destructor Destroy; override;
  function Compare(Range: TSynCustomHighlighterRange): integer; virtual;
  function Add(ABlockType: Pointer = nil; IncreaseLevel: Boolean = True):
      TSynCustomCodeFoldBlock; virtual;
  procedure Pop(DecreaseLevel: Boolean = True); virtual;
  function MaxFoldLevel: Integer; virtual;
  procedure Clear; virtual;
  procedure Assign(Src: TSynCustomHighlighterRange); virtual;
  procedure WriteDebugReport;
  property FoldRoot: TSynCustomCodeFoldBlock read FTop write FTop;
public
  property RangeType: Pointer read FRangeType write FRangeType;
  property CodeFoldStackSize: integer read FCodeFoldStackSize;
  property MinimumCodeFoldBlockLevel: integer
    read FMinimumCodeFoldBlockLevel write FMinimumCodeFoldBlockLevel;
  property Top: TSynCustomCodeFoldBlock read FTop;
end;
```

Este objeto, funciona a manera de pila con respecto a los plegados. El método Add(), agrega un nuevo elemento y Pop(), extrae el último elemento.

El método Pop(), tiene una protección a intentar eliminar un plegado que no existe. Así que podría ejecutarse sin temor a desbordar a "CodeFoldRange".

Para ver el tamaño de la pila, se puede llamar en todo momento a "CodeFoldStackSize". Este contador solo se actualizará cuando se llame a StartCodeFoldBlock()con el parámetro "IncreaseLevel" en TRUE o a EndCodeFoldBlock() con el parámetro "DecreaseLevel" en TRUE, como es la forma por defecto.

2.5.5 Habilitar y deshabilitar bloques de plegado

Una característica deseable de la funcionalidad de plegado, es que se pueda habilitar o deshabilitar ciertos bloques de plegado.

En la sección anterior, vimos cómo se puede deshabilitar bloques de plegado, en el resaltador, usando los parámetros "IncreaseLevel" y "DecreaseLevel". Ahora veremos cómo usar unas estructuras

predefinidas en el resaltador, para facilitar el manejo de la habilitación/inhabilitación de los bloques de plegado.

La clase "TSynCustomFoldHighlighter", incluye métodos y propiedades para manejar lo que se llama "Configuraciones de plegado":

```
  TSynCustomFoldHighlighter = class(TSynCustomHighlighter)
  protected
    // Fold Config
    FFoldConfig: Array of TSynCustomFoldConfig;
    function GetFoldConfig(Index: Integer): TSynCustomFoldConfig; virtual;
    procedure      SetFoldConfig(Index:       Integer;       const       AValue:
TSynCustomFoldConfig); virtual;
    function GetFoldConfigCount: Integer; virtual;
    function GetFoldConfigInternalCount: Integer; virtual;
    function   GetFoldConfigInstance(Index:   Integer):   TSynCustomFoldConfig;
virtual;
    procedure InitFoldConfig;
    procedure DestroyFoldConfig;
    procedure DoFoldConfigChanged(Sender: TObject); virtual;
  private
...
  protected
...
  public
    property FoldConfig[Index: Integer]: TSynCustomFoldConfig
      read GetFoldConfig write SetFoldConfig;
    property FoldConfigCount: Integer read GetFoldConfigCount;

  end;
```

Estas propiedades nos proporcionan una estructura interna, para el almacenamiento de las propiedades de configuración del plegado, y expone además, las propiedades FoldConfig y FoldConfigCount, para acceder a estas propiedades.

Es posible usar cualquier otra estructura personalizada, pero TSynCustomFoldHighlighter, incluye esta clase y los métodos y propiedades necesarios para su manejo, de forma que nos ahorra el trabajo de manejar configuraciones de los rangos de plegado. Además al estar incluidas en la misma clase TSynCustomFoldHighlighter, son siempre accesibles desde los resaltadores.

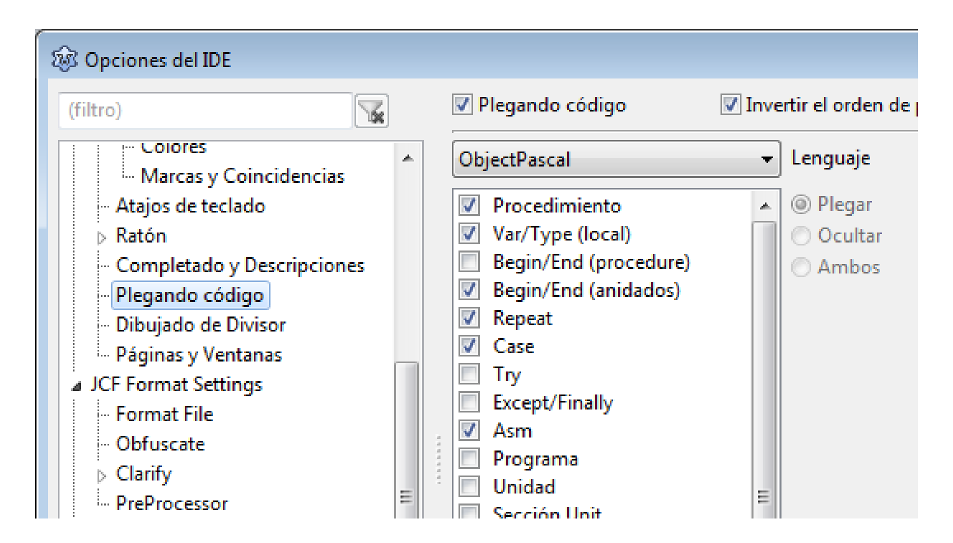
Las configuraciones de un rango de plegado se guardan en el arreglo FFoldConfig. Un objeto FoldConfig, es una instancia de TSynCustomFoldConfig:

```
  TSynCustomFoldConfig = class(TPersistent)
  private
    FEnabled: Boolean;
    FFoldActions: TSynFoldActions;
    FModes: TSynCustomFoldConfigModes;
    FOnChange: TNotifyEvent;
    FSupportedModes: TSynCustomFoldConfigModes;
```

```
...
  published
    property Enabled: Boolean read FEnabled write SetFEnabled;
    property Modes:  TSynCustomFoldConfigModes  read  FModes  write  SetModes
default [fmFold];
  end;
```

La propiedad más importante es quizá “Enabled”, porque nos permite decidir cuándo un rango de plegado está o no habilitado.

Esta estructura es una base con las propiedades básicas para un rango de plegado. Para tener una idea de cómo se utilizan, se puede ir al entorno de Lazarus, y seleccionar el menú “Herramientas>Opciones>Editor>Plegado de Código>”:



En este caso, se están mostrando los bloques de plegado para el resaltador de Pascal de Lazarus. Cuando se activa o desactiva alguna de estas casillas, se está cambiando la propiedad “Enabled”, de uno de los objetos “`TSynCustomFoldConfig`” que existen en el resaltador.

En la implementación de la mayoría de resaltadores con plegado en Lazarus, se verá que, para decidir hacer o no visible el plegado, se consulta primero al arreglo FFoldConfig[].

```
function  TSynLFMSyn.StartLfmCodeFoldBlock(ABlockType:  TLfmCodeFoldBlockType):
TSynCustomCodeFoldBlock;
var
  FoldBlock: Boolean;
  p: PtrInt;
begin
  FoldBlock :=  FFoldConfig[ord(ABlockType)].Enabled;
  p := 0;
  if not FoldBlock then
    p := PtrInt(CountLfmCodeFoldBlockOffset);
  Result := StartCodeFoldBlock(p + Pointer(PtrInt(ABlockType)), FoldBlock);
end;

procedure TSynLFMSyn.EndLfmCodeFoldBlock;
var
```

```
  DecreaseLevel: Boolean;
begin
  DecreaseLevel := TopCodeFoldBlockType < CountLfmCodeFoldBlockOffset;
  EndCodeFoldBlock(DecreaseLevel);
end;
```

En este código, se usa el truco de dar a "p", un valor desplazado del valor original, cuando se usa un plegado no visible, en StartLfmCodeFoldBlock(). De modo que, luego se pueda recuperar su valor original y de paso saber si el plegado era visible o no, en EndLfmCodeFoldBlock().

Este método es una forma de codificar información adicional en un enumerado (disfrazado de puntero). Como consecuencia, se debe también aplicar esta decodificación, sobrescribiendo TopLfmCodeFoldBlockType():

```
function TSynLFMSyn.TopLfmCodeFoldBlockType(DownIndex: Integer):
TLfmCodeFoldBlockType;
var
  p: Pointer;
begin
  p := TopCodeFoldBlockType(DownIndex);
  if p >= CountLfmCodeFoldBlockOffset then
    p := p - PtrUInt(CountLfmCodeFoldBlockOffset);
  Result := TLfmCodeFoldBlockType(PtrUInt(p));
end;
```

Por norma, se espera que existan tantos objetos TSynCustomFoldConfig, como bloques de plegado distintos. Así por ejemplo, en el resaltador SynLFMSyn, existen los siguientes bloques:

```
  TLfmCodeFoldBlockType = (
    cfbtLfmObject,      // object, inherited, inline
    cfbtLfmList,        // <>
    cfbtLfmItem,         // Item
    cfbtLfmNone
    );
```

Por lo tanto, se deben sobreescribir los métodos GetFoldConfigInternalCount() y GetFoldConfigInstance() para que se creen las configuraciones, necesarias:

```
function TSynLFMSyn.GetFoldConfigInternalCount: Integer;
begin
  Result := ord(high(TLfmCodeFoldBlockType)) -
ord(low(TLfmCodeFoldBlockType)) + 1;
end;

function TSynLFMSyn.GetFoldConfigInstance(Index: Integer):
TSynCustomFoldConfig;
begin
  Result := inherited GetFoldConfigInstance(Index);
  Result.Enabled := True;
end;
```

Estos métodos se ejecutan al inicio, para crear las estructuras de configuración en FFoldConfig[].

2.6 La clase TSynCustomFoldHighlighter

Como hemos visto, todos los resaltadores que deseen implementar plegado de código, deben derivar de la clase TSynCustomFoldHighlighter en lugar de TSynCustomHighlighter.

Se podría decir que, la clase TSynCustomFoldHighlighter es un descendiente de TSynCustomHighlighter, que simplemente agrega la funcionalidad de plegado de código.

Como es de esperar la declaración de la clase, agrega la información necesaria para el plegado de código y unos métodos útiles para obtener información sobre el plegado.

2.6.1 El plegado a bajo Nivel

Para entender mejor cómo se implementa el plegado en la clase, describiremos la mecánica de trabajo a bajo nivel.

Consideremos un código típico en Pascal, con plegado:

```
procedure primero;
begin
  max := 0;
  if a>b then
  begin
    max := a;
  end
  else
  begin
    max := b;
  end;
end;
```

En este código se ve que todo el procedimiento se encuentra dentro de un bloque de plegado. El cuerpo del procedimiento abre otro bloque de plegado. Dentro, la estructura IF, abre dos bloques de plegado: el cuerpo del IF y el cuerpo del ELSE.

Como es común, podemos ahorrar unas líneas uniendo algunas palabras claves en la misma línea:

```
procedure primero;
begin
  max := 0;
  if a>b then begin
    max := a;
  end else begin
    max := b;
  end;
end;
```

En este código se tiene que el cuerpo del IF termina en la misma línea en donde empieza el cuerpo del ELSE, por ello, se aprecia una especia de "solapamiento" de los bloques, pero no es así. En realidad se podría decir que el cuerpo del IF "termina", después de la palabra reservada END, y que el cuerpo el ELSE, empieza con la palabra reservada BEGIN.

Como los bloques de plegado, se pueden anidar, entonces estamos introduciendo el concepto de "nivel". Si no hemos abierto ningún bloque de plegado, diremos que estamos en el nivel cero. Cuando se abre un bloque de plegado, diremos que estamos en el nivel 1 y así sucesivamente. Es por ello que comúnmente, estaremos hablando de "niveles de anidamiento" dentro de un bloque de plegado.

Al igual que en el manejo de rangos, por cuestiones de economía y simplicidad, la información relativa a bloques de plegado, se almacena por cada línea, como el estado del resaltador al terminar de explorar cada línea.

Para poder procesar correctamente, el plegado de código, la clase TSynCustomFoldHighlighter, guarda dos valores por cada línea:

- EndLevel.- Es el nivel de anidamiento de bloques al final de la línea.
- MinLevel.- Es el nivel de anidamiento mínimo en cualquier parte de la línea.

Estas variables, y otras más se guardan como parte de la información que el resaltador almacena en cada línea del editor.

Un ejercicio visual nos ayudará a entender cómo se procesa el plegado de código dentro de TSynCustomFoldHighlighter:

	EndLevel	MinLevel
`procedure primero;`	1	0
`begin`	2	1
`  max := 0;`	2	2
`  if a>b then begin`	3	2
`    max := a;`	3	3
`  end else begin`	3	2
`    max := b;`	3	3
`  end;`	2	2
`end;`	0	0

En la primera línea el nivel mínimo es 0, porque antes de la palabra reservada PROCEDURE, no se ha abierto todavía ningún bloque.

En la segunda línea se abre un nuevo bloque con la palabra reservada BEGIN. Para fines de plegado de código, no es tan importante saber si el bloque empieza antes o después de BEGIN.

Las demás líneas siguen la misma lógica.

Esta información es todo lo que se necesita el editor para mostrar las marcas de plegado en el panel lateral. Cada vez que se encuentra que EndLevel > MinLevel, significa que se ha abierto uno o más bloques de plegado, y se debe mostrar la marca de bloque. Si EndLevel, en la línea anterior es mayor que MinLevel, en una línea, significa que el bloque de la línea anterior se ha cerrado. Esta información permite realizar exploraciones rápidas en las líneas para obtener información sobre los bloques de plegado.

Desde luego que TSynCustomFoldHighlighter, maneja mayor información para lidiar con los bloques, pero para fines de visualización de marcas, esta información sería suficiente, y es parte de la información que se almacena en cada línea del editor.

Puede parecer que se pierde información del plegado si es que se abre y cierra un bloque, dentro de una misma línea, pero hay que considerar que para fines de plegado de código, ese bloque resultaría intrascendente.

Para ver los valores EndLevel y MinLevel, se puede acceder a las propiedades:

```
function TSynCustomFoldHighlighter.FoldBlockEndLevel(ALineIndex: TLineIdx;
  const AFilter: TSynFoldBlockFilter): integer;
function TSynCustomFoldHighlighter.FoldBlockMinLevel(ALineIndex: TLineIdx;
  const AFilter: TSynFoldBlockFilter): integer;
```

Solo es necesario pasar el número de línea requerido (iniciando en cero). El “campoAFilter” no se usa, así que se debe dejar en NIL.

2.6.2 CurrentLines[] y CurrentRanges[]

Estas propiedades de TSynCustomFoldHighlighter, devuelven la misma información que devolvería TSynCustomHighlighter, más o menos.

CurrentLines[], es exactamente igual y permite acceder a las líneas del editor actual. Hasta ahí todo funciona bien.

Sin embargo CurrentRanges[], ha cambiado.

El hecho de que TSynCustomFoldHighlighter, no solo almacene información de Rangos, hace necesario que se tenga una forma de agregar información adicional a cada línea del editor.

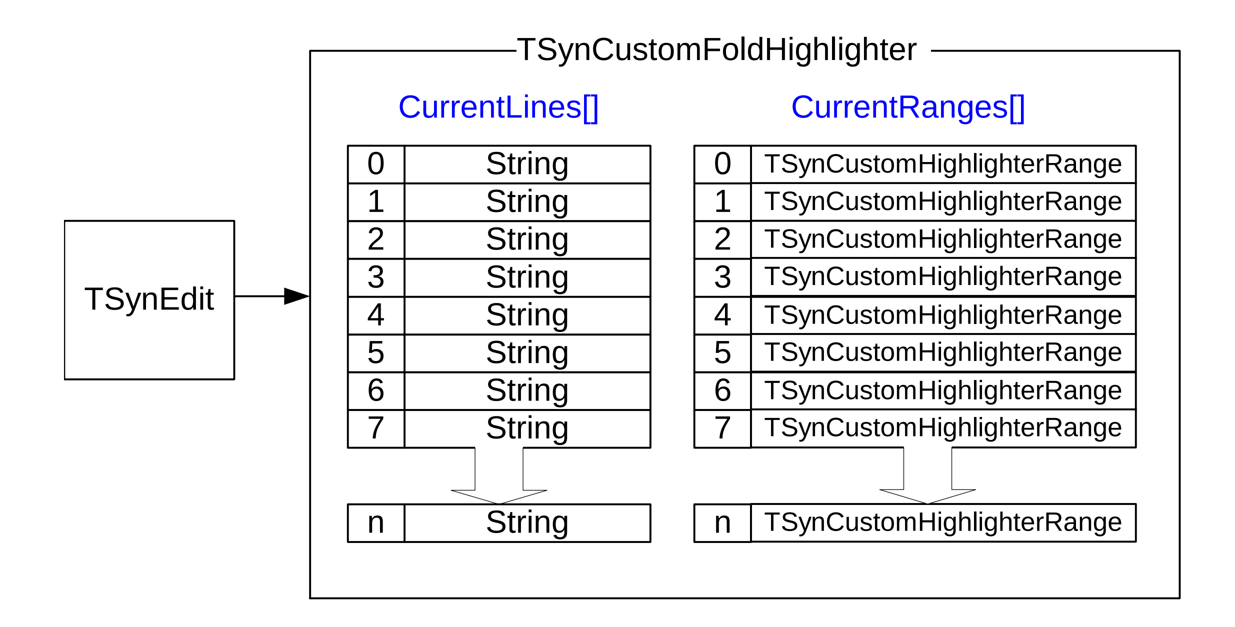
El manejo de bloques de plegado, requiere información adicional, así que se ha creado una estructura especial que permita empaquetar la información de rango (el mismo puntero de siempre), y adicionalmente otros campos más.

Por ello es que existe la clase TSynCustomHighlighterRange, definida de la siguiente forma:

```
TSynCustomHighlighterRange = class
private
  FCodeFoldStackSize: integer; // EndLevel
  FMinimumCodeFoldBlockLevel: integer;
  FRangeType: Pointer;
  FTop: TSynCustomCodeFoldBlock;
public
  constructor Create(Template: TSynCustomHighlighterRange); virtual;
  destructor Destroy; override;
  function Compare(Range: TSynCustomHighlighterRange): integer; virtual;
  function Add(ABlockType: Pointer = nil; IncreaseLevel: Boolean = True):
      TSynCustomCodeFoldBlock; virtual;
  procedure Pop(DecreaseLevel: Boolean = True); virtual;
  function MaxFoldLevel: Integer; virtual;
  procedure Clear; virtual;
  procedure Assign(Src: TSynCustomHighlighterRange); virtual;
  procedure WriteDebugReport;
  property FoldRoot: TSynCustomCodeFoldBlock read FTop write FTop;
public
  property RangeType: Pointer read FRangeType write FRangeType;
  property CodeFoldStackSize: integer read FCodeFoldStackSize;
  property MinimumCodeFoldBlockLevel: integer
    read FMinimumCodeFoldBlockLevel write FMinimumCodeFoldBlockLevel;
  property Top: TSynCustomCodeFoldBlock read FTop;
end;
```

Esto significa que ahora TSynCustomFoldHighlighter, cuando se lea CurrentRanges[] para una línea, no leeremos directamente el valor del rango, sino que obtendremos la referencia a un objeto TSynCustomHighlighterRange, con información adicional (mucha información adicional), necesaria para el manejo de los bloques de plegado.

El siguiente diagrama aclara mejor la situación:



La información de rango, pasa a ser ahora un humilde campo más de este nuevo objeto. El campo es RangeType. Es por ello que, en un resaltador con plegado, cuando se implementan los métodos:

TSynFacilSyn.GetRange: Pointer;

TSynFacilSyn.SetRange(Value: Pointer);

, se debe trabajar con el campo "CodeFoldRange.RangeType", en lugar del valor directo de la función.

CurrentRanges[] todavía almacena punteros, como lo hace en TSynCustomHighlighter, pero ahora estos punteros son referencias a objetos de tipo TSynCustomFoldHighlighterRange.

Así que podemos aplicar la conversión de tipos cuando leamos CurrentRanges[]:

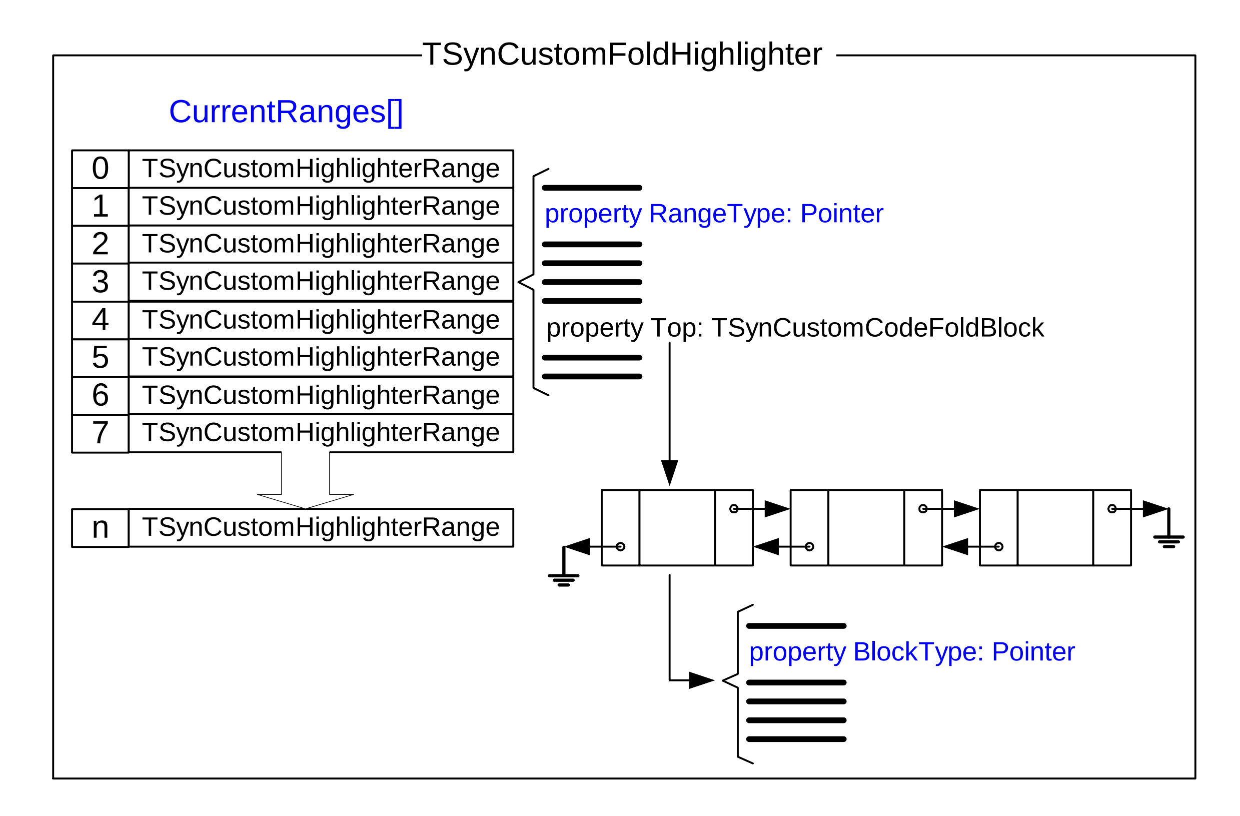
```
p := TSynCustomHighlighterRange(CurrentRanges[SynEdit1.CaretY-1])
```

Se podría decir que las propiedades más importantes de un objeto TSynCustomFoldHighlighterRange, son RangeType y Top.

La propiedad Top, es una referencia a un objeto TSynCustomCodeFoldBlock, que representa al último Bloque de Plegado que se ha abierto, al final de la línea correspondiente de CurrentRanges[].

Un objeto TSynCustomCodeFoldBlock, no es el bloque de plegado exactamente (el que abrimos con StartCodeFoldBlock()), sino que es un objeto que funciona como el nodo de una lista enlazada.

Esto es así porque todos los bloques abiertos, con StartCodeFoldBlock(), se almacenan en una lista enlazada, y no en una pila LIFO, como se podría esperar.



Los métodos Add() y Pop() de TSynCustomHighlighterRange, permiten agregar o quitar nodos de la lista.

Cada nodo de la lista es un objeto de tipo TSynCustomCodeFoldBlock, que tiene la siguiente definición:

```
  TSynCustomCodeFoldBlock = class
  private
    FBlockType: Pointer;
    FParent, FChildren: TSynCustomCodeFoldBlock;
    FRight, FLeft: TSynCustomCodeFoldBlock;
    FBalance: Integer;
    function GetChild(ABlockType: Pointer): TSynCustomCodeFoldBlock;
  protected
    function GetOrCreateSibling(ABlockType: Pointer):
TSynCustomCodeFoldBlock;
    property Right: TSynCustomCodeFoldBlock read FRight;
    property Left: TSynCustomCodeFoldBlock read FLeft;
    property Children: TSynCustomCodeFoldBlock read FChildren;
  public
    destructor Destroy; override;
    procedure WriteDebugReport;
  public
    procedure InitRootBlockType(AType: Pointer);
    property BlockType: Pointer read FBlockType;
    property Parent: TSynCustomCodeFoldBlock read FParent;
    property Child[ABlockType: Pointer]: TSynCustomCodeFoldBlock read
GetChild;
  end;
```

Los enlaces a los otros nodos, se encuentran en las propiedades “Parent” y “Children”.

El dato más importante que contiene este nodo es BlockType, que es la referencia el bloque de plegado que indicamos cuando usamos StartCodeFoldBlock(). Existe un BlockType por cada nodo, es decir por cada bloque abierto.

Como sucede con el RangeType, el puntero BlockType, puede hacer referencia a un objeto real o ser un entero o enumerado codificado en un dato de tipo puntero. Esto dependerá de cómo se implemente el resaltador que haga uso del plegado de código.

2.6.3 Algunos Métodos y Propiedades

Para el manejo del plegado, TSynCustomFoldHighlighter tiene diversos métodos disponibles, que pueden ser accedidos desde dentro y fuera de la clase.

El método FoldEndLine(), con la declaración:

```
function     TSynCustomFoldHighlighter.FoldEndLine(ALineIndex,      FoldIndex:
Integer): integer;
```

, permite devolver la línea final del bloque, al final de la línea ALineIndex (que empieza en 0 para la primera línea). El segundo parámetro, permite especificar el nivel del bloque a usar. Para el bloque de mayor nivel, se debe dejar en cero.

Consideremos el siguiente código:

```
1 procedure primero;
2 begin
3   max := 0;
4   if a>b then begin
5     max := a;
6   end else begin
7     max := b;
8   end;
9 end;
```

Si aplicamos la función FoldEndLine(1,0), obtendremos 8, porque el bloque de mayor nivel abierto, al final de la línea 2, termina en la línea 9.

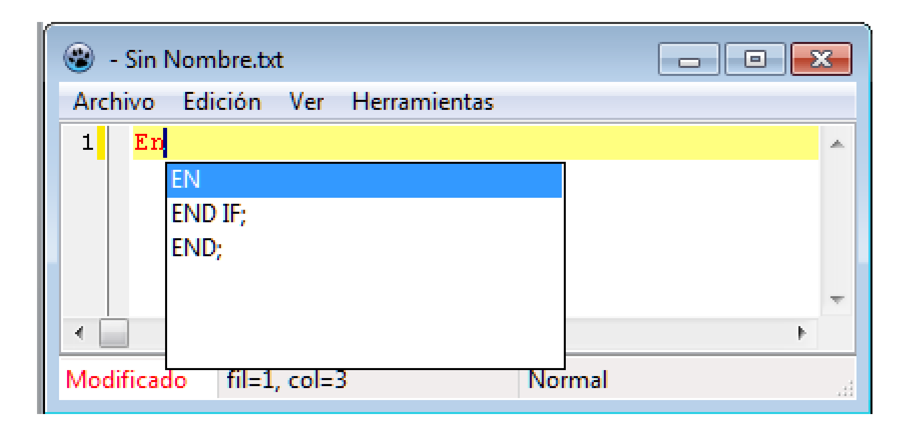
El método FoldLineLength(), devuelve el número de líneas del bloque abierto al final de la línea indicada. Tiene los mismos parámetros que FoldEndLine():

```
function    TSynCustomFoldHighlighter.FoldLineLength(ALineIndex,    FoldIndex:
Integer): integer;
```

De la misma forma, ALineIndex, empieza en cero para la línea 1.

2.7 Autocompletado

El autocompletado es la característica que tiene SynEdit que permite mostrar, mientras se escribe, una lista de palabras que podemos elegir para completar el texto faltante en la palabra actual. De esta forma nos ahorrarnos el trabajo de tener que escribir toda la palabra.



El autocompletado en SynEdit, no está tan desarrollado, como la característica de plegado o de resaltado de sintaxis. Existen funcionalidades básicas, pero cumple su cometido.

Como varias otras características de SynEdit, es posible implementar autocompletado usando componentes ("TSynCompletion"), de la barra de componentes, o se puede utilizar código para crearlo.

El autocompletado, se ha definido para que tenga el siguiente *modus operandi*:

1. Está asociado siempre a un editor de tipo SynEdit. Esta asociación se puede hacer en diseño, o por código.

2. Se activa cada vez, que se pulsa la combinación de teclas asociadas al completado. La combinación más común usada es <Ctrl> + <espacio>.

3. Al activarse, debe aparecer una ventana, en la posición del cursor, con una lista de palabras (o frases), de las que se debe seleccionar una de ellas.

4. Al mostrarse la ventana de selección, se toma la palabra actual, la que esta antes del cursor, como palabra de trabajo. Esta es la palabra que será reemplazada.

5. SI al mostrarse la ventana de selección, esta solo tiene una opción, se procederá a remplazar automáticamente la palabra de trabajo con esta única opción y se cerrará la ventana de selección.

6. Cuando está mostrada la ventana de selección, se puede seguir escribiendo sobre el editor, pero la ventana de selección tomará el control de algunas teclas, como las direccionales. Además no se permitirá eliminar la palabra de trabajo, porque debe tener al menos un caracter.

7. En la lista mostrada, para desplazarse por las opciones se pueden usar las teclas direccionales, <Pag.Arriba>, <Pag.Abajo>, <Inicio>, o <Fin>. Estas teclas dejarán de actuar sobre el editor, mientras esté mostrada la lista de opciones.

8. Para seleccionar una de las opciones se debe pulsar la tecla <enter>, <espacio>, o algún símbolo como '+', o '-'.

9. Al seleccionar una de las opciones de la lista, se reemplazará la palabra actual, por la seleccionada, y desaparecerá la ventana de selección.

10. SI no se quiere seleccionar alguna opción de la ventana de opciones, se debe pulsar <escape>, para hacer desaparecer la ventana de opciones y continuar la edición normal.

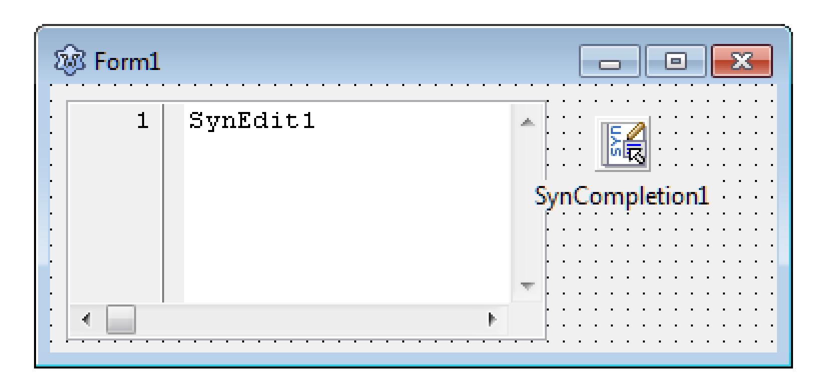
11. Solo se puede hacer desaparecer, la ventana de opciones, seleccionando alguno de sus ítems o pulsando <escape>.

Este es el comportamiento normal de la funcionalidad de autocompletado. Para cambiar este comportamiento, se debe manipular las propiedades del componente de completado (TSynCompletion), tanto por la ventana de diseño, o por código.

Para modificaciones mayores, se debe hacer enteramente por código.

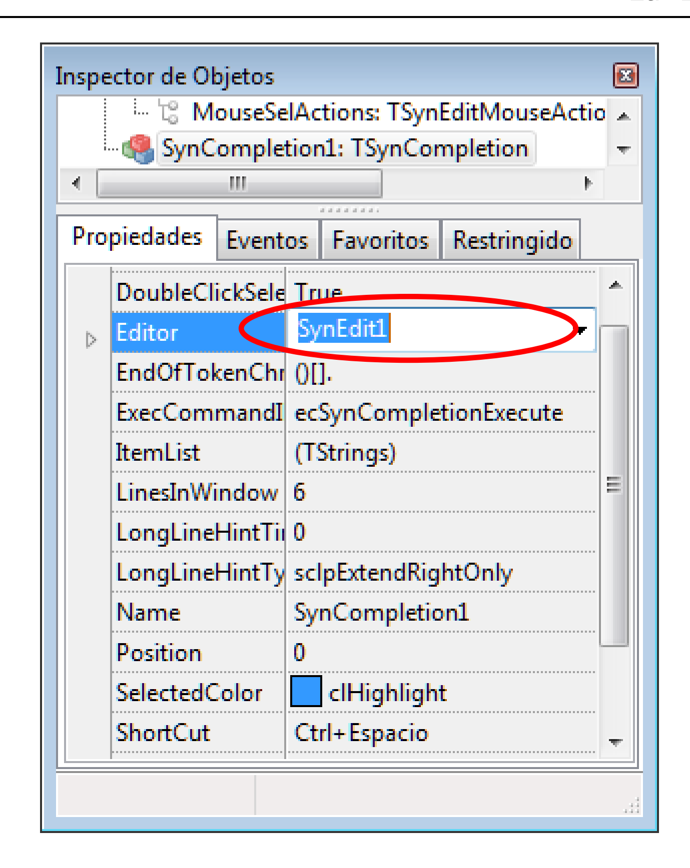
2.7.1 Autocompletado usando componente

El procedimiento es sencillo. Se agrega el editor TSynEdit, al formulario, como es natural, y luego se agrega también el componente "TSynCompletion":



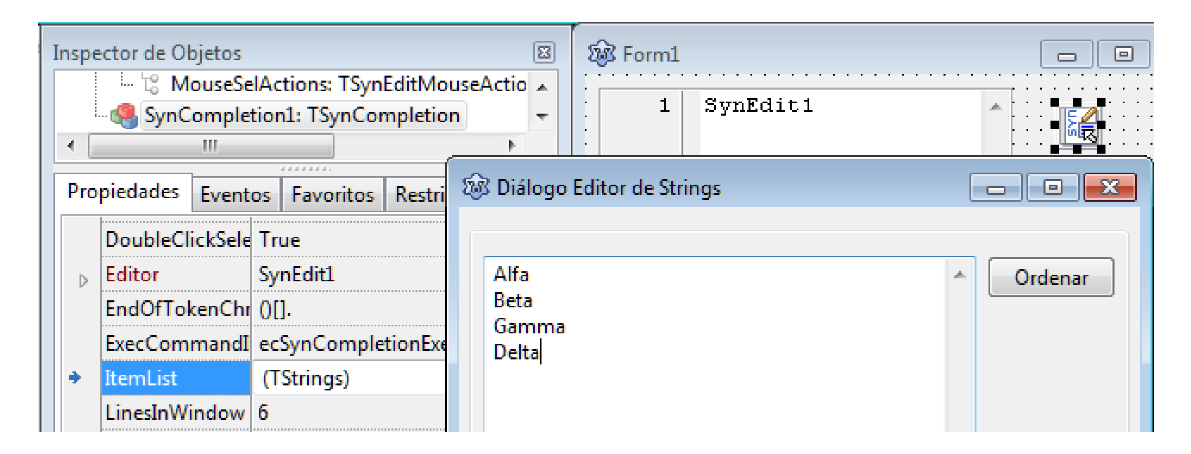
El componente "TSynCompletion", se encuentra en la paleta de componentes, donde se encuentra también "TSynEdit".

Una vez que tengamos los componentes de trabajo, se requiere asociar "SynCompletion1" a "SynEdit1". Para ello, se configura en el explorador de objetos, la propiedad "Editor" de "SynCompletion1", para que apunte a "SynEdit1":



Observar que la propiedad por defecto para “ShortCut”, es “Ctrl + Espacio”, lo que significa que esta combinación será la que active la lista desplegable del autocompletado.

Una vez asociada esta propiedad, ya estará lista la funcionalidad para trabajar en el editor. Pero todavía se necesita agregar la lista de palabras que se mostrará en el menú desplegable. Para ello podemos modificar directamente desde el Inspector de objetos, la propiedad “ItemList”:



Ahora, cuando se ejecute el programa, y se pulse la combinación <Ctrl> + <espacio>, aparecerá una lista desplegable con las palabras indicadas, que nos permitirá reemplazar la palabra actual con una de las palabras de la lista.

Este método de trabajo, es sencillo y rápido de implementar, pero no permite opciones más elaboradas de trabajo como filtrar la lista de acuerdo a la palabra que se está escribiendo. Para esta y otras funcionalidades, es necesario trabajar con código.

2.7.2 Autocompletado usando código

Para el autocompletado, se debe crear un objeto de la clase “TSynCompletion”:

```
  MenuContex: TSynCompletion;    //menú contextual
```

La inicialización es sencilla:

```
    MenuContex:=TSynCompletion.Create(Self0);    //crea menú
    MenuContex.Editor:=ed;      //asigna editor
    MenuContex.ShortCut:=Menus.ShortCut(VK_SPACE, [ssCtrl]);  //asigna atajo
de teclado
```

En condiciones normales, la ventana de completado, solo se activará cuando se presionen, la combinación de teclas del atajo. Si se desea mostrar la ventana, en cualquier otro momento, se puede usar el método execute:

El siguiente procedimiento, muestra la ventana de completado “TSynCompletion1”, para el editor “ed”:

```
procedure Mostrar;
var p:TPoint;
begin
    P := Point(ed.CaretXPix,ed.CaretYPix + ed.LineHeight);
    P.X:=Max(0,Min(P.X, ed.ClientWidth - TSynCompletion1.Width));
    P := ed.ClientToScreen(p);
    //Abre menú contextual. Solo se mostrará si tiene elementos.
    SynCompletion1.Execute('', p.x, p.y);
end;
```

El cálculo de p.x y p.y, se hace para que la ventana aparezca en la posición del cursor. Es importante notar que la ventana, solo se mostrará, si “TSynCompletion1”, tiene elementos en su lista.

Para agregar elementos en “TSynCompletion1”, se debe usar la propiedad “ItemList”, que es un “StringList”:

```
   SynCompletion1.ItemList.add('Alfa');
   SynCompletion1.ItemList.add('Beta');
   SynCompletion1.ItemList.add('Gamma');
```

Las cadenas que se agregan, son las que aparecerán al mostrarse la ventana del “TSynCompletion”.

Si la lista de palabras va a ser fija, entonces se puede llenar una sola vez al inicio del programa. Si la lista depende de la palabra actual, o de otra información, entonces se puede hacer uso de uno de los eventos de “TSynCompletion”. El evento en cuestión es “OnExecute”. Este evento se ejecutará antes de abrir la lista de palabras para el completado. Aquí se puede aprovechar para elegir la lista de palabras a

llenar. Como ayuda adicional, se puede usar la cadena "SynCompletion1.CurrentString". Aquí se guarda el identificador que se encuentra antes del cursor, es decir la palabra de trabajo

El evento "OnExecute", se lanzará independientemente de si se activó la lista por combinación de teclas o usando el método "Execute". Pero solo cuando se activa con combinación de teclas se actualiza automáticamente la cadena "CurrentString". Cuando se usa "Execute", el primer parámetro es el valor inicial que se asigna a "CurrentString".

Con cada tecla pulsada, mientras está activa la ventana de selección, se actualiza la cadena "CurrentString".

Otro evento importante es "OnSearchPosition". Este evento es llamado cada vez que se pulsa una tecla mientras se tiene la ventana del "TSynCompletion" abierta. Como parámetro entrega un entero que indica el elemento seleccionado. El primer elemento es el 0. Este valor se puede cambiar desde dentro del evento, para elegir otro elemento a seleccionar. Si no usa este evento, por defecto, se selecciona el elemento que coincida con "CurrentString".

La propiedad "CurrentString", indica la palabra actual la que está detrás del cursor. No importa si hay caracteres delante del cursor, "CurrentString" solo considera los caracteres anteriores, hasta encontrar un espacio. Esta propiedad solo se actualiza, al invocar el "TSynCompletion" desde teclado. Si se abre con "Execute()", "CurrentString" iniciará vacía.

3 APÉNDICE

3.1 Algoritmos para la implementación de resaltadores.

Se pueden implementar de diversas formas. El objetivo principal es que sea de respuesta rápida y se enfocan sobretodo en la comparación rápida de cadenas. Los algoritmos que más se han ensayado son:

3.1.1 Algoritmo basado en el uso de funciones Hash

Este es el algoritmo que se ha usado para implementar la mayoría de los resaltadores pre-definidos que vienen con Lazarus. Es de procesamiento rápido, pero tiene una implementación complicada.

Trabaja con una tabla de funciones "fProcTable[]", que se usa para direccionar a cada carácter, de acuerdo a su tipo, a una función específica que se encargará de identificar el token analizado.

El llenado de esta tabla se hace antes de usar el resaltador, en la función `MakeMethodTables()`:

```
procedure TSynPerlSyn.MakeMethodTables;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      #0: fProcTable[I] := @NullProc;
      #1..#9,#11,#12,#14..#32: fProcTable[I] := @SpaceProc;
      #10: fProcTable[I] := @LFProc;
      #13: fProcTable[I] := @CRProc;
      ':': fProcTable[I] := @ColonProc;
      '#': fProcTable[I] := @CommentProc;
      '=': fProcTable[I] := @EqualProc;
      '>': fProcTable[I] := @GreaterProc;
      '<': fProcTable[I] := @LowerProc;
      '0'..'9', '.': fProcTable[I] := @NumberProc;
      '$', 'A'..'Z', 'a'..'z', '_': fProcTable[I] := @IdentProc;
      '-': fProcTable[I] := @MinusProc;
      '+': fProcTable[I] := @PlusProc;
      '/': fProcTable[I] := @SlashProc;
      '*': fProcTable[I] := @StarProc;
      #34: fProcTable[I] := @StringInterpProc;
      #39: fProcTable[I] := @StringLiteralProc;
    else
      fProcTable[I] := @UnknownProc;
    end;
end;
```

Aquí los caracteres alfabéticos, se consideran siempre como inicio de identificadores y se hacen apuntar a la función IdentProc() que se encargará de extraer el identificador y verificar si se trata de una palabra clave.

Para detectar identificadores claves sigue el siguiente proceso:

Se extrae el token que corresponde al identificador y se calcula el resultado de su función hash[13]. Con este valor se direcciona una tabla de funciones, llamada fIdentFuncTable[], que debe tener entradas habilitadas solo en los valores de la función hash que correspondan a palabras claves.

La tabla fIdentFuncTable[] se llena en el método InitIdent, y debe hacerse antes de usar el resaltador:

```
procedure TSynPerlSyn.InitIdent;
var
  I: Integer;
begin
  for I := 0 to 2167 do
    Case I of
      109: fIdentFuncTable[I] := @Func109;
      113: fIdentFuncTable[I] :=@func113;
      196: fIdentFuncTable[I] :=@func196;
      201: fIdentFuncTable[I] :=@func201;
      204: fIdentFuncTable[I] :=@func204;
      207: fIdentFuncTable[I] :=@func207;
      209: fIdentFuncTable[I] :=@func209;
      211: fIdentFuncTable[I] :=@func211;
      230: fIdentFuncTable[I] :=@func230;
      ...
    else
      fIdentFuncTable[I] := @AltFunc;
    end;
end;
```

Las funciones direccionadas son de la forma “funcXXX”, donde XXX es el valor “hash“ que corresponde. Para un grupo de palabras claves, solo existen un número de valores “hash” en el rango.

Las otras entradas solo devuelven el valor “tkIdentifier” (a través de AltFunc), que indica que se trata de un simple identificador.

---

[13] Este valor se obtiene creando una tabla que asigna un valor a cada letra del alfabeto ingles (usualmente en la tabla mHashTable[], y aprovechando para llenarla en la función “MakeIdentTable”). Con esta tabla se calcula el valor que le corresponde a cada palabra clave, sumando el valor de cada letra en el identificador. El valor obtenido suele depender de las letras del identificador y de su tamaño, pero usualmente no supera a 200 en una sintaxis normal. Este valor no es único para cada palabra, ya que varias palabras distintas pueden compartir el mismo valor, pero permite categorizar de manera efectiva a los identificadores para restringir la búsqueda a un grupo mucho menor.

Si existen varias palabras claves que tienen el mismo valor "hash", se implementa una comparación adicional, dentro de la función que corresponda:

```
function TSynPerlSyn.Func230: TtkTokenKind;
begin
  if KeyComp('tr') then Result := tkKey else
    if KeyComp('my') then Result := tkKey else Result := tkIdentifier;
end;
```

3.1.2 Algoritmo basado en el Primer carácter como prefijo.

Este algoritmo es el que describimos en este documento y se basa en usar el primer carácter de un identificador como prefijo, para acelerar la detección de palabras claves.

También utiliza una tabla de funciones "fProcTable[]", que se usa para direccionar a cada carácter, de acuerdo a su tipo, a una función específica que se encargará de identificar el token analizado. La diferencia está en que esta tabla apuntará a una función específica para cada carácter alfabético:

```
procedure TSynMiColorSF.MakeMethodTables;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      '-'     : fProcTable[I] := @ProcMinus;
      '/'     : fProcTable[I] := @ProcSlash;
      #39     : fProcTable[I] := @ProcString;
      '"'     : fProcTable[I] := @ProcString2;
      '0'..'9': fProcTable[I] := @ProcNumber;
      'A','a': fProcTable[I] := @ProcA;
      'B','b': fProcTable[I] := @ProcB;
      'C','c': fProcTable[I] := @ProcC;
      'D','d': fProcTable[I] := @ProcD;
      ...
      'Z','z': fProcTable[I] := @ProcZ;
      '_'     : fProcTable[I] := @ProcUnder;
      '$'     : fProcTable[I] := @ProcMacro;
      #13     : fProcTable[I] := @ProcCR;
      #10     : fProcTable[I] := @ProcLF;
      #0      : fProcTable[I] := @ProcNull;
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
end;
```

Luego cada función de tipo ProcA() o ProcB(), hace directamente las comparaciones para identificar las palabras claves:

```
procedure TSynMiColorSF.ProcA;
begin
  while Identifiers[fLine[Run]] do inc(Run);
  fStringLen := Run - fTokenPos - 1;  //calcula tamaño - 1
  fToIdent := Fline + fTokenPos + 1;  //puntero al identificador + 1
  if KeyComp('nd') then fTokenID := tkKey else
  if KeyComp('rray') then fTokenID := tkKey else
    fTokenID := tkIdentifier;  //identificador común
end;
```

Como ya se ha identificado el primer carácter del identificador, la comparación se hace con un carácter menos, lo que acelera la comparación.

3.1.3 Comparación entre algoritmos

Para verificar el rendimiento de ambos algoritmos, he realizado dos grupos de comparaciones. Uno de ellos usando el resaltador PHP de Lazarus 1.0.12 y el otro usando el resaltador Perl.

Ambos resaltadores (implementados con el algoritmo hash) fueron comparados en velocidad con un resaltador implementado con el algoritmo de Primer Carácter como Prefijo.

Estos fueron los resultados:

Resaltador PHP (con algoritmo hash):	1.1seg, 1.1 seg, 1.1 seg.
Resaltador PHP (con algoritmo de prefijo):	1.0 seg, 1.0 seg, 1.0 seg.

Resaltador Perl (con algoritmo hash):	1.84 seg, 1.82 seg, 1.83 seg
Resaltador Perl (con algoritmo de prefijo):	1.68 seg, 1.68 seg, 1.68 seg.

Las pruebas muestran los tiempos que toma procesar un archivo completo un número determinado de veces. La prueba con Perl, realizaba 5000 exploraciones sencillas a un archivo. La rutina de exploración tenía esta forma:

```
procedure ExplorarArchivo(lineas: TStringList; hlt: TSynCustomHighlighter);
//Explora un archivo usando el resaltador indicado.
var
  p: PChar;
  tam: integer;
  lin : string;
begin
  for lin in lineas do begin
    hlt.SetLine(lin,1);
    while not hlt.GetEol do begin
      hlt.Next;
      hlt.GetTokenEx(p,tam);
```

```
      hlt.GetTokenAttribute;
    end;
  end;
end;
```

Ambas pruebas, se hicieron tomando un archivo fuente en PHP y Perl, como ejemplos. El de Perl tenía 427 líneas.

Después de realizadas las pruebas, se concluye que el algoritmo de prefijos es por lo general más rápido que el de funciones "hash". Es posible que haya casos particulares, en los que esto no sea cierto, pero siempre se podrá mejorar la implementación del algoritmo de prefijos para que sea comparativamente más rápido.

Considerando otros criterios para la comparación, se puede elabora la siguiente tabla:

CRITERIO	ALGORITMO DE FUNCIONES HASH	ALGORITMO DEL PRIMER CARÁCTER COMO PREFIJO
Velocidad	Más lento en la mayoría de casos	Más rápido en la mayoría de casos
Tamaño de código	Mayor, por el hecho de requerir una tabla de funciones adicional y una cantidad grande de funciones.	Menor, porque solo usa una tabla de funciones y comparaciones sencillas.
Optimizable	Difícil de optimizar. Requiere mejorar la función hash, pero puede complicar el tiempo de procesamiento.	Más fácil de optimizar. Se puede agregar más niveles en forma de árbol de prefijos o se puede mejorar las comparaciones.
Legibilidad	No muy legible. Difícil de seguir el código.	Es más legible.
Mantenimiento.	Difícil de mantener. Agregar nuevas palabras claves implica hacer un nuevo cálculo de su función hash.	Fácil de mantener. Agregar nuevas palabras claves solo implica ponerlas en la función que corresponda.
Posibilidad de usar archivos externos de sintaxis.	Muy pocas. Su estructura inherente complica hacerlo dinámico.	Más manejable. Al ordenar sus identificadores, permite adaptarlo mejor al uso de archivos externos de sintaxis.

De acuerdo a la comparación hecha, NO SE RECOMIENDA, implementar resaltadores de sintaxis con el algoritmo de funciones Hash, que se ha usado en SynEdit.

3.1.4 Criterios de optimización

He insistido repetidas veces en la necesidad de usar algoritmos de respuesta rápida en la implementación de resaltadores.

Aquí quiero indicar algunos puntos a tener en cuenta en la implementación de algoritmos para los resaltadores, considerando que estamos usando el compilador Free Pascal en su versión 2.6.2. Algunos de estos criterios, podrían no ser ciertos en otros compiladores o inclusive en otras versiones de Free Pascal.

El primer punto que debemos considerar es que las cadenas de tipo PChar, son generalmente más rápidas que las de tipo String, ya que se manejan como punteros.

Otra consideración con respecto a punteros es que, si se tiene una cadena de tipo PChar, declarada así:

fLine: PChar;

La forma, más rápida de acceder a un caracter es: fLine[i]

La forma: (fLine+i)^ es notoriamente más lenta (se ha estimado un 7% más lenta).

En los resaltadores, es común tener que avanzar un puntero mientras el carácter apuntado pertenezca a un conjunto válido de caracteres.

Para comparar rápidamente si un carácter se encuentra en un conjunto de caracteres, se analizan tres métodos:

<u>Matriz de valores booleanos</u>

Un algoritmo típico sería:

```
var CharsIdentif: array[#0..#255] of ByteBool;
...
  while CharsIdentif[fLine[posFin]] do
    inc(posFin);
```

Aquí se supone que el arreglo CharsIdentif[], ha sido inicializado para los caracteres que se desean considerar como válidos.

<u>Comparar con Conjuntos</u>

Un algortimo típico sería:

```
var letras : set of char;
...
 while fLine[posFin] in letras do
   inc(posFin);
```

Aquí se supone que el conjunto letras, se ha iniciado con los caracteres que se desean considerar como válidos.

<u>Comparar con Case ... Of</u>

Un método rápido de comparación se puede implementar también con la siguiente estructura:

```
while true do begin
  case fLine[posFin] of
  '$','_','0'..'9','a'..'z','A'..'Z': inc(posFin);
  else break;  //sale
  end
end;
```

Una tabla comparativa muestra, las velocidades de respuesta de estos tres métodos, ensayados en Windows con arquitectura x86 de 32 bits:

MÉTODO	TIEMPO
Matriz de valores booleanos	39
Comparación con conjuntos	48
Comparar con Case ... Of	45

Se puede ver que la comparación usando matriz de booleanos, es cerca de un 20% más rápida que la comparación usando conjuntos y cerca de 13% más rápida que usando Case ..Of.

Se ensayó también una comparación usando una matriz de números enteros, en lugar de valores booleanos, obteniéndose un retardo mayor. Usar enteros es cerca de 1% o 2% más lento.

ÍNDICE